Universidad Autónoma de Barcelona

Escuela Técnica Superior de Ingeniería

Departamento de Informática

CoScheduling Cooperativo: una propuesta de coscheduling orientada a clusters no dedicados multiprogramados

Tesis presentada por Francesc Giné de Sola para optar al grado de Doctor por la Universidad Autónoma de Barcelona

Barcelona, Abril de 2004

CoScheduling Cooperativo: una propuesta de coscheduling orientada a clusters no dedicados multiprogramados

Memoria presentada por Francesc Giné de Sola para optar al grado de Doctor por la Universidad Autónoma de Barcelona. Trabajo realizado en el Departamento de Informática de la Escuela Técnica Superior de Ingeniería de la Universidad Autónoma de Barcelona, bajo la dirección del Dr. Porfidio Hernández Budé.

Barcelona, Abril de 2004

Vo. Bo. Director de Tesis

Fdo. Porfidio Hernández Budé

Per a tu, la teva llum

il·lumina el meu caminar.

Agradecimientos

Deseo expresar mi agradecimiento al Dr. Porfidio Hernández Budé por su magistral dirección y constante dedicación a la supervisión del trabajo.

Al Dr. Emilio Luque, junto con todas las personas de la Unidad de Arquitectura de Computadores y Sistemas Operativos de la UAB, por habernos integrado, tanto a mi como a mis compañeros de área de la Universitat de Lleida, en su grupo de investigación. Por habernos hecho sentir uno más del grupo, tanto a nivel de trabajo como personal.

A las personas que han colaborado de forma más directa en el desarrollo de los trabajos relacionados con esta tesis, Francesc Solsona y Mauricio Hanzich. A todos ellos les quiero agradecer todas aquellas aportaciones que han contribuido al desarrollo de este trabajo de tesis.

Un especial agradecimiento para todos mis compañeros de área de la Universitat de Lleida que me han tenido que soportar durante todos estos largos años. A Concepció Roig, por su compañerismo y apoyo siempre sincero. A Fernando Cores, por su optimismo y sentido del humor. A Fernando Guirado, Josep M. Erill, Josep M. Solà y Josep L. Lérida, por su constante ayuda en todos aquellos momentos, que no han sido pocos, donde la oscuridad me invadía.

No puedo olvidarme de todos aquellos compañeros y amigos de la EPS que me han brindado toda su amistad y optimismo, escuchando mis rollos en nuestras tertulias de café matinales y que a pesar de las mismas, todavía no me han retirado la palabra.

Y como no, a toda mi familia. A mis padres, hermanos, a mi mujer, Eulàlia, y a los pequeños, Mariona y Francesc, porque a pesar de mi importante dedicación en los últimos años a la tesis, siempre han sabido percibir que ellos, para mi, son lo más importante.

Prólogo

En la década de los 90, con la expansión y desarrollo de las workstations (o PCs), diferentes estudios realizados sobre el grado de utilización de los recursos de cómputo (CPU y memoria) de una workstation, pusieron de manifiesto que un elevado porcentaje de los mismos estan infrautilizados. La posibilidad de utilizar esta potencia de cálculo disponible en una red de PCs (cluster/NOW:"*Network Of Workstations*") para la ejecución de aplicaciones distribuidas con un rendimiento equivalente a un MPP ("*Massively Parallel Processor*"), sin perturbar en exceso el trabajo del usuario local de cada workstation, fue objeto de estudio por parte de la comunidad científica.

En la práctica, una serie de características hacen que un entorno NOW difiera substancialmente de un MPP. Entre las más representativas se pueden citar:

- *Red de interconexión.* En una NOW los elementos de procesamiento (PE) están interconectados mediante una red LAN ("*Local Area Network*") comercial, mientras que en un MPP los PEs están conectados por una red propietaria de alta velocidad.

- *Dedicación.* Un entorno NOW puede ser no dedicado y heterogéneo, frente a un MPP que es homogéneo y totalmente dedicado a la computación paralela.

- *Sintonización.* Un MPP está sintonizado para la ejecución de aplicaciones paralelas, mientras que un entorno NOW está preparado para sacar el máximo rendimiento a las aplicaciones ejecutadas en cada nodo por el usuario local, caracterizadas por un alto grado de interactividad.

Estas divergencias entre un MPP y un entorno NOW conllevan que las políticas de planificación orientadas a los MPPs no sean válidas para entornos NOW, hecho que ha motivado el desarrollo de nuevas técnicas de planificación orientadas a entornos NOW.

Las nuevas técnicas desarrolladas con el objetivo de compartir los recursos de cómputo de una NOW entre los usuarios locales, de cada workstation, y los usuarios de aplicaciones distribuidas, ejecutadas a lo largo de toda la NOW, se agrupan en dos áreas diferentes: técnicas de *espacio compartido* y técnicas de *tiempo compartido*. Diferentes estudios revelan que las técnicas de tiempo compartido permiten utilizar de un modo más eficiente estos recursos disponibles para la ejecución de las aplicaciones distribuidas. Sin embargo, un problema intrínseco de las técnicas de tiempo compartido es el modo de

garantizar la coplanificación de aquellas tareas distribuidas que se comunican entre si, denominadas *tareas cooperantes*; siendo este problema conocido como *coscheduling*. El principal objetivo del coscheduling es minimizar el tiempo de espera de las tareas distribuidas ante los eventos de comunicación y sincronización, de modo que todas las tareas de una misma aplicación distribuida puedan progresar coordinadamente a lo largo del cluster.

Las *técnicas tradicionales de coscheduling* se pueden clasificar en dos grupos diferentes: las técnicas de *coscheduling con control explícito*, basadas en planificar simultáneamente todas las tareas de una misma aplicación distribuida mediante un cambio de contexto global a lo largo del cluster, y las *técnicas con control implícito*, donde cada nodo planifica las tareas distribuidas, residentes en el mismo, de acuerdo con sus necesidades de comunicación y sincronización. De este modo, las técnicas con control implícito predicen la necesidad de coscheduling entre tareas cooperantes a partir de la información local obtenida implícitamente ante la ocurrencia de ciertos eventos de comunicación y sincronización. La flexibilidad asociada al mecanismo de coscheduling empleado por las técnicas con control implícito permite que estas técnicas se puedan adaptar dinámicamente a la alta variabilidad de recursos disponibles, que caracteriza un entorno NOW no dedicado. Este hecho ha motivado el desarrollo de diferentes propuestas de coscheduling con control implícito orientadas a entornos cluster no dedicados.

Sin embargo, las técnicas de coscheduling con control implícito basadas en el análisis de los eventos de comunicación no pueden garantizar, por si solas, un control sobre el rendimiento de las aplicaciones del usuario local de cada estación, así como de las aplicaciones distribuidas ejecutadas a lo largo de la NOW. Este hecho ha comportado que en las diferentes implementaciones realizadas de las mismas, en entornos NOW no dedicados, se haya restringido el grado de multiprogramación paralela (MPL) a uno. Esta restricción en el grado de multiprogramación paralela conlleva que se desperdicien una gran cantidad de recursos de cómputo disponibles a lo largo de un cluster. Ésto es debido a que durante esta última década las necesidades de los usuarios locales no han evolucionado al mismo ritmo que la tecnología. Ambas tendencias comportan que la distancia entre las prestaciones ofrecidas por los equipos informáticos respecto a las necesidades reales de los usuarios aumente, de modo que los recursos de cómputo no utilizados por los usuarios locales de un cluster se incrementan a lo largo del tiempo. Este progresivo aumento de los recursos de cómputo disponibles plantea la posibilidad de incrementar el grado de multiprogramación de las aplicaciones paralelas en un entorno NOW no dedicado.

La ejecución de múltiples aplicaciones paralelas sobre un cluster no dedicado comporta ampliar el espectro de eventos a analizar por las técnicas de

coscheduling subyacentes, con el objetivo de garantizar el progreso de las diferentes aplicaciones distribuidas, así como de asegurar el rendimiento de las aplicaciones locales dentro de unos márgenes aceptables por el usuario local. Aspectos tales como los requerimientos de memoria, de CPU y de comunicación, la longitud del quantum, el rendimiento de la cache, la presencia de usuarios locales en determinados nodos del cluster, el balanceo de recursos de cómputo a lo largo del cluster y el grado de heterogeneidad hardware del cluster pueden tener una gran influencia en el rendimiento de este tipo de entornos y, por consiguiente, deben ser tenidos en cuenta por las correspondientes políticas de coscheduling aplicadas en cada uno de los nodos del cluster.

Este nuevo marco de trabajo impone, en nuestro modo de ver, un replanteamiento del problema del coscheduling de aplicaciones distribuidas. El fin del coscheduling no solamente se debe restringir a decidir *cuándo* deben ser asignados los recursos de cómputo a las aplicaciones distribuidas, finalidad de las técnicas de coscheduling tradicionales, si no también *cuántos* recursos deben asignarse a cada aplicación. Este doble propósito nos ha llevado a desarrollar una nueva propuesta de coscheduling, denominada *CoScheduling Cooperativo* (*CS*C), orientada a la coordinación de múltiples aplicaciones paralelas en un entorno cluster no dedicado.

CSC, siguiendo el principio de coscheduling con control implícito, es un sistema totalmente distribuido. CSC, a diferencia de las propuestas de coscheduling con control implícito tradicionales, gestiona los recursos de cómputo de cada nodo tanto en función de la ocurrencia de determinados eventos locales (de memoria, de CPU, de comunicación y de actividad del usuario local), como de la recepción de aquellos eventos ocurridos en nodos remotos y que han modificado los recursos asociados a los procesos cooperantes.

El análisis de estos eventos permite a CSC adaptar los recursos de cómputo del cluster a las necesidades de ambos tipos de usuarios; el usuario local, caracterizado por unos elevados requerimientos de interactividad, y el usuario paralelo, en el cual priman los requerimientos de cómputo y de comunicación. A nivel local, CSC prioriza la asignación tanto de la CPU como de la memoria al usuario local, garantizando los tiempos de respuesta de sus aplicaciones dentro de unos margenes aceptables para el mismo. El resto de recursos locales son asignados por CSC atendiendo tanto a las necesidades de las tareas distribuidas residentes en dicho nodo, como a los requerimientos de las tareas cooperantes ejecutándose en nodos remotos. De este modo, CSC consigue balancear los recursos asignados a las tareas que forman una misma aplicación distribuida a lo largo del cluster.

En esta tesis, se describe como CSC es capaz de alcanzar los objetivos citados anteriormente, evaluando y comparando su rendimiento con respecto

a las técnicas de coscheduling tradicionales, tanto por medio de la simulación como de la implementación en un entorno cluster real. Con este fin, la presente memoria se organiza en los siguientes capítulos:

- Capítulo 1. Se enmarca el problema del coscheduling y se describen las alternativas clásicas, abarcando tanto las técnicas de coscheduling con control explícito como las de control implícito. A partir de la descripción de dichas técnicas, se justifica la necesidad de la nueva propuesta de coscheduling.

- Capítulo 2. A partir de la descripción del modelo de cluster, sobre el cual se desarrollan nuestros estudios, en este capítulo se evalúan, tanto por medio de la simulación como de la experimentación en un entorno cluster, aquellos aspectos que, atendiendo tanto a referencias de la literatura científica como a nuestra propias intuiciones, influyen en el rendimiento de las tareas distribuidas y de las locales.

- Capítulo 3. A partir de los resultados obtenidos en el capítulo precedente, en este capítulo se desarrolla nuestra nueva propuesta de *Co-Scheduling Cooperativo*, con una descripción detallada, a nivel algorítmico, de cada uno de los módulos que la constituyen, así como de la interacción entre los mismos. Asimismo, por medio de la simulación, se muestra como CSC aumenta el rendimiento de las técnicas de coscheduling tradicionales debido a su capacidad de gestionar la ejecución de múltiples aplicaciones paralelas sin perturbar, en exceso, el trabajo de los usuarios locales presentes a lo largo del cluster.

- Capítulo 4. A partir de la descripción de las características de un entorno PVM-Linux, en este capítulo se explica la implementación de CSC sobre dicho entorno.

- Capítulo 5. La efectividad de CSC es analizada para un conjunto de benchmarks distribuidos sobre un cluster de PCs, con diferentes grados de utilización por parte de los usuarios locales. El rendimiento de CSC, tanto en lo que se refiere al punto de vista del usuario local como del usuario de las aplicaciones distribuidas, es evaluado con respecto a diferentes políticas de coscheduling tradicionales presentes en la literatura. Estas pruebas se realizan tanto en un entorno cluster controlado como en un entorno productivo con usuarios locales reales.

- Capítulo 6. Finalmente, se presentan las principales conclusiones que se derivan del presente trabajo, junto con las contribuciones a que ha

dado lugar el desarrollo del mismo. Asimismo, se indican las líneas que quedan abiertas a partir de los resultados obtenidos.

Estos capítulos se complementan con los siguientes apéndices:

- Apéndice A: Se describe el entorno de simulación implementado específicamente para el desarrollo de esta tesis.
- Apéndice B: Se muestran todos aquellos resultados experimentales adicionales que se complementan con la experimentación desarrollada a lo largo de los capítulos 2 y 5.
- Apéndice C: Se explican las distintas llamadas a sistema implementadas para la gestión de CSC.
- Apéndice D: Se describen las herramientas de monitorización y gestión implementadas durante el desarrollo de esta tesis.
- Apéndice E: Se explica el benchmark utilizado para simular los requerimientos de un usuario local, así como los parámetros utilizados para la calibración del mismo.

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Coscheduling en un Entorno Cluster no Dedicado

Las *estaciones de trabajo* (ws o PCs[1]) son sistemas diseñados para proporcionar un alto rendimiento al usuario local, en aquellas aplicaciones que requieran una gran interactividad y normalmente un pequeño volumen de computo. Por el contrario, los *supercomputadores* están orientados a obtener el mejor rendimiento posible en aplicaciones con un elevado volumen de cálculo, a costa de un elevado precio.

El creciente volumen de ventas en el mercado de los PCs ha provocado que los fabricantes amorticen rápidamente los gastos en investigación, lo que ha comportado un incremento anual en la relación rendimiento/precio de aproximadamente un 80 %, mientras que en el reducido mercado de los supercomputadores dicho incremento ha sido de aproximadamente un 20 % [ACP+95]. Este hecho, junto con la introducción de nuevas tecnologías asociadas a las redes locales de alta velocidad [Nea95, All96], han permitido que la potencia de cómputo de una red de PCs (cluster o NOW), debidamente configurada, pueda competir con la de los supercomputadores [top02].

Hoy en día, clusters de considerable tamaño están disponibles en los centros de investigación, universidades e incluso en numerosas empresas de pequeño y mediano tamaño. Recientes estudios realizados sobre el porcentaje de utilización de dichos clusters [ML91, ACP+95], muestran que incluso durante las horas laborables, más del 50 % de workstations están desocupadas; mientras que por las noches y fines de semana se alcanza un porcentaje de desocupación alrededor del 90 % [ACP+95, AES97]. Esta baja tasa de utilización ha creado grandes expectativas ante la posibilidad de utilizar estos recursos de cómputo disponibles para ejecutar grandes aplicaciones parale-

[1]En este contexto los términos workstation (ws) o PC se utilizan como sinónimos.

las, sin perturbar, en exceso, el trabajo del usuario local; entendiendo como *usuario local* a aquella persona que trabaja en modo interactivo con una máquina, ya sea su legítimo propietario o bien un usuario temporal de una máquina perteneciente a una institución (p.e. un estudiante trabajando en el laboratorio de usuarios de su respectiva facultad). De este modo, se han abierto las puertas a la computación paralela en entornos cluster no dedicados.

En la práctica, un cluster no dedicado se caracteriza por ser un entorno muy dinámico, donde los recursos de cómputo asignados a cada aplicación fluctúan a lo largo del tiempo [ACP+95, Ryu01, DZ97, AES97]. Estas características comportan que las políticas de planificación clásicas utilizadas en los supercomputadores no sean válidas en los entornos cluster. Este hecho ha despertado el interés de la comunidad científica; de manera que durante esta última década, se han propuesto nuevos métodos de planificación sobre entornos cluster no dedicados. Muchas de estas nuevas propuestas pueden enmarcarse en las tres áreas siguientes:

1. Técnicas basadas en servicios de ejecución remota. Entre los sistemas basados en esta técnica podríamos destacar, *Piranha* [CFGK95] *y Process Server* [Hag88].

2. Técnicas basadas en la migración de tareas. En esta área se enmarcan *MOSIX* [BL98], *Condor* [LLM88], *Stealth* [KB93] y *Sprite* [OCD+88].

3. Técnicas de planificación basadas en la coplanificación de tareas (*coscheduling*), tales como *Gang Scheduling* [FR92] o *coscheduling Dinámico* [SPWC98].

Tanto las técnicas de ejecución remota como las técnicas basadas en migración de procesos han sido creadas para ejecutar las aplicaciones paralelas solamente cuando el usuario local no está utilizando su máquina. Esto significa que los procesos paralelos, ejecutándose en una determinada máquina, son parados y migrados tan pronto como el usuario local de dicha máquina reanuda su trabajo. Esta manera de proceder provoca que una gran cantidad de recursos de cómputo, que podrían ser utilizados para la ejecución paralela, sean desperdiciados. Ésto es debido a que incluso cuando el usuario está trabajando activamente en su máquina, el uso de recursos es muy bajo. De acuerdo con el estudio realizado por Ryu [Ryu01], el perfil mayoritario entre los usuarios de una ws en una aula de estudiantes se corresponde a una persona que utiliza su máquina como una herramienta ofimática, es decir para tareas tales como el procesamiento de textos, edición de hojas de cálculo, envío y recepción de correo, etc... En este tipo de tareas, el usuario

gasta la mayor parte del tiempo en pensar y escribir [Nie00], lo que comporta que los recursos de cómputo de una ws estén siendo totalmente utilizados en ocasiones excepcionales. El uso de las técnicas de coscheduling permite que en una misma máquina puedan coexistir tanto aplicaciones locales como paralelas, característica que conlleva un uso más eficiente de estos recursos. Siguiendo este razonamiento, nuestra investigación se ha centrado en el uso de técnicas de coscheduling para ejecutar múltiples aplicaciones paralelas sin perturbar el trabajo del usuario local.

Con el objetivo de enmarcar nuestro trabajo, una descripción de todos aquellos componentes que integran un entorno cluster se realiza en la sección 1.1. A continuación, en la sección 1.2 se introducen las diferentes técnicas de planificación en entornos cluster, con un especial énfasis en aquellas técnicas orientadas a clusters no dedicados. En la sección 1.3 se explican las diferentes aportaciones presentes en la literatura en el campo del coscheduling. A partir de los trabajos precedentes en el área del coscheduling, en la sección 1.4 se justifica la necesidad de una nueva propuesta de coscheduling orientada a clusters no dedicados multiprogramados, y se explican los principales objetivos asociados con dicha propuesta.

1.1. Elementos de un Cluster no Dedicado

Con objeto de enmarcar los entornos cluster dentro de la clasificación de máquinas paralelas, en la figura 1.1 se muestra la taxonomía desarrollada por Tanembaum [Tan99] y completada, en sus dos últimos niveles, por la clasificación de las NOW realizada por Buyya [Buy99].

El primer nivel de la taxonomía está basado en la clasificación de Flynn [Fly72]. Flynn usa el concepto de *flujos de instrucciones* y *datos* como base para clasificar los diferentes tipos de arquitecturas paralelas. Por ejemplo, *SISD* es el acrónimo de "*Single Instruction stream and Single Data stream*" y corresponde a la máquina clásica de Von Neumann. Nuestro interés se centra en la categoría *MIMD* ("*Multiple Instruction stream and Multiple Data stream*"), que corresponde a una máquina con múltiples elementos de procesamiento (PEs) independientes que ejecutan diferentes procesos en paralelo utilizando distintos conjuntos de datos. Dentro de esta categoría se engloban tanto los *multiprocesadores* (MIMD de memoria compartida) como los *multicomputadores* (MIMD de memoria distribuida). La principal diferencia entre ambos estriba en el hecho de que un multicomputador está formado por un conjunto de elementos de procesamiento relativamente autónomos, en los que cada CPU dispone de su propia memoria principal, mientras que en un multiprocesador existe una memoria común compartida por un conjunto de

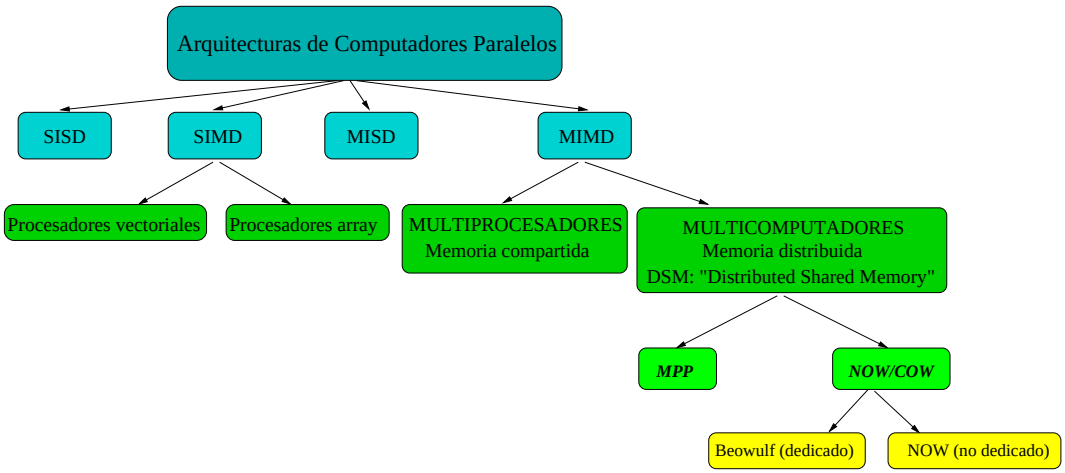

Figure 1.1: Taxonomía de arquitecturas paralelas.

procesadores, los cuales están bajo el control de un mismo sistema operativo.

Los multicomputadores, a su vez, se dividen en MPPs ("*Massively Parallel Processors*") y NOWs/COWs ("*Networks/Clusters*[2] *of Workstations*"). Un MPP se caracteriza por ser un supercomputador formado por centenares de elementos de procesamiento, los cuales son interconectados por una red propietaria de alta velocidad. A título de ejemplo, podríamos citar los bien conocidos supercomputadores *Cray T3E* y el *IBM SP/2*.

Los clusters o NOWs consisten en un conjunto de workstations o PCs, de propósito general, conectados por una red local comercial (LAN). La principal diferencia entre un MPP y una NOW, aparte de su precio, es la velocidad de la red de interconexión de los MPP, aunque cabe decir que, cada vez, esta diferencia se va reduciendo. De hecho, en la última década, el uso de clusters como una alternativa a los MPP se ha popularizado, básicamente por los motivos que se enumeran a continuación:

- La potencia de cómputo de las ws comerciales se ha visto incrementada espectacularmente en los últimos años, de hecho su velocidad es doblada aproximadamente cada 24 meses. Además del incremento de la potencia de cálculo, las ws actuales presentan una nada despreciable capacidad de memoria y disco.

- Las redes de área local tradicionales de tipo bus, como ethernet e hiperbus, no son capaces de soportar el ancho de banda y la latencia necesarias para poder ejecutar aplicaciones paralelas de propósito general.

[2]Los términos NOW o cluster se utilizarán como sinónimos a lo largo del texto.

Por esta razón, históricamente, las NOWs solamente se han utilizado para ejecutar aplicaciones paralelas de granularidad gruesa. La reciente introducción de LANs de alta velocidad como Fast Ehternet o FDDI [Tan99] y, sobretodo, el desarrollo de LANs conmutada como ATM [BCS93] y Myrinet [Nea95], con unos anchos de banda superiores a los 100 Mbps y latencias inferiores a *1ms*, han permitido el uso de las NOW para la ejecución de aplicaciones paralelas de propósito general.

- La baja utilización, por parte de los usuarios locales, de los recursos de cómputo presentes en una NOW.

- La estandarización de muchas de las herramientas y utilidades usadas por las aplicaciones paralelas en entornos cluster, en contraste con los sistemas propietarios utilizados por los MPP. Como ejemplo de estos estándares, podríamos citar la librería de paso de mensajes MPI ("*Message Passing Interface*") [For02] o el lenguaje paralelo HPF ("*High Performance Fortran*") [Kea94].

- El aumento en un cluster, tanto del número de nodos como de la potencia de cómputo de un nodo, es relativamente fácil y barato; hecho que favorece la escalabilidad de un cluster.

Dentro de los clusters, se distinguen dos tipos diferentes: los "*Beowulf*" y los *clusters no dedicados*. La principal diferencia entre ambos estriba en el hecho de que los Beowulf son sistemas dedicados y sintonizados para la ejecución de aplicaciones paralelas, mientras que en un cluster no dedicado o simplemente NOW, se ejecutan tanto aplicaciones paralelas como las aplicaciones locales asociadas a los propietarios o usuarios locales de cada una de las máquinas que integran el cluster. Nuestro trabajo está orientado hacia estas plataformas no dedicadas.

La figura 1.2 muestra la arquitectura típica de un sistema cluster no dedicado [Buy99]. En las siguientes secciones se procederá a describir las principales características de cada uno de los componentes que la integran.

1.1.1. Red de interconexión

De acuerdo con los datos mostrados en la figura 1.3, el ancho de banda de las redes locales se está adaptando a las necesidades de los usuarios actuales, con una progresiva substitución de las redes con ancho de banda a 100Mbps por las de 1000Mbps, como son *Gigabit Ethernet* o *Myrinet*.

Asimismo, la comunidad científica ha trabajado en el desarrollo de nuevos protocolos de redes que disminuyan el *overhead* asociado a los mismos. Uno

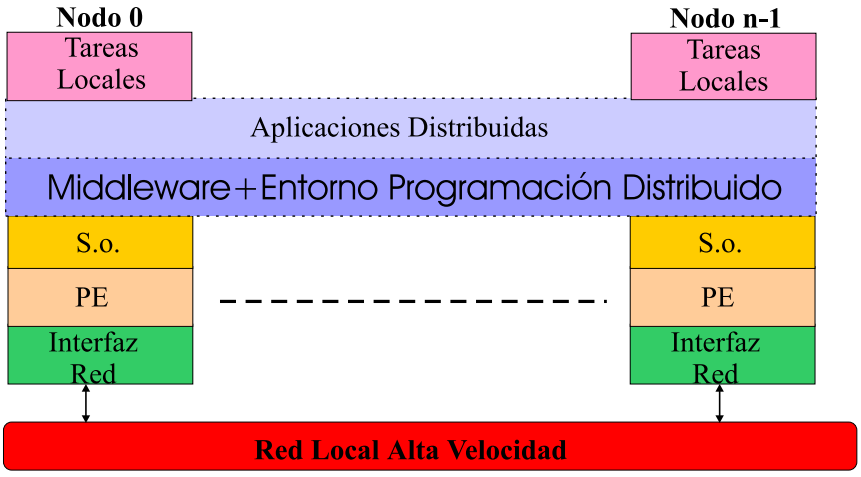

Figura 1.2: Arquitectura de un cluster no dedicado.

de los problemas asociados con el clásico estándar *TCP(UDP)/IP* es el elevado overhead que introduce en la transmisión y recepción de mensajes con respecto a la baja latencia de transmisión asociada a los anchos de banda actuales. Este elevado overhead es debido a los diferentes niveles de procesamiento y copias temporales en memoria que sufre cada mensaje enviado y recibido en cada una de las capas que integran el protocolo TCP(UDP)/IP. En este sentido, cabe destacar el nuevo protocolo de redes denominado *Active Messsages* [ECGS92], el cual reduce considerablemente dichos overheads.

Dado que nuestros esfuerzos se centran en la propuesta de nuevas técnicas de coscheduling de tareas, independientemente del tipo de red de interconexión subyacente, no haremos mayor hincapie en el estudio de la influencia, en el rendimiento global del sistema, del tipo de red a utilizar.

1.1.2. PE: Elementos de procesamiento

Básicamente, los tres principales componentes que integran cada uno de los elementos de procesamiento (PE) que componen un cluster son tres: el procesador, la memoria y la entrada/salida (I/O). La interacción entre estos tres subsistemas, así como la capacidad de cada uno de ellos, influyen directamente en el rendimiento de la computación paralela en entornos NOW no dedicados [AES97, BF00, FFPF03, ADV+95].

Durante las dos últimas décadas, la arquitectura de los microprocesadores ha experimentado progresos espectaculares (por ejemplo, RISC, CISC, VLIW), de manera que los microprocesadores disponibles actualmente en un PC co-

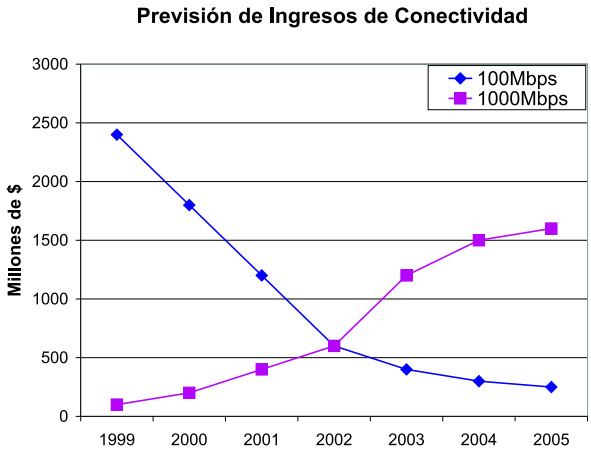

Figure 1.3: Tendencia de las redes de interconexión [Cor02a].

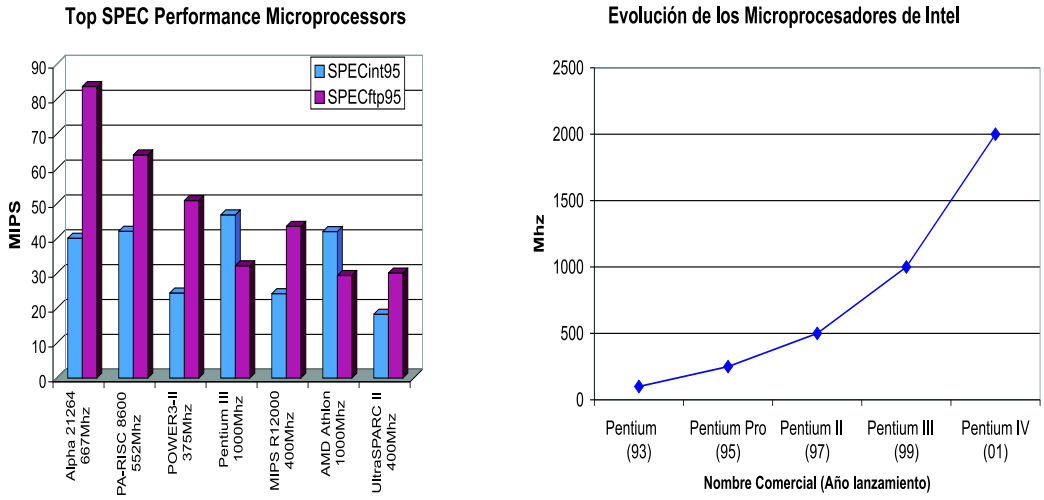

Figura 1.4: (Izquierda) Rendimiento SPEC de diferentes procesadores comerciales [Cor03b]. (Derecha) Evolución de los procesadores Intel [Cor02b].

mercial ofrecen una potencia de cómputo casi similar a la de los procesadores que albergan los supercomputadores. La figura 1.4(izq.) muestra una comparación entre el rendimiento alcanzado por diferentes procesadores comerciales al ejecutar la conocida suite de benchmarks SPEC95 [Cor03b].

Este progreso se refleja en la evolución seguida durante esta última déca-

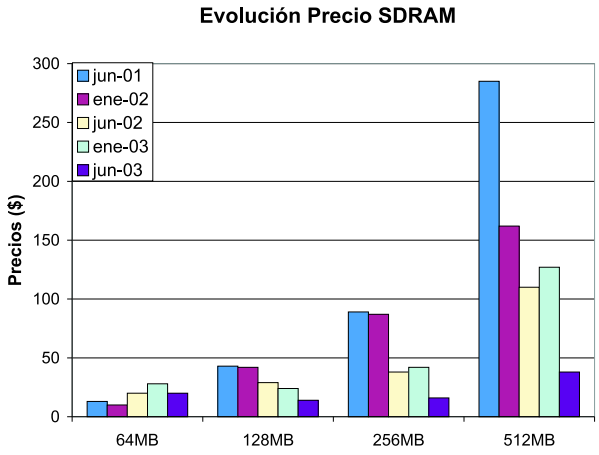

Figura 1.5: Evolución de la relación tamaño/precio de los chips de memoria SDRAM [Cor03a].

da por los procesadores Intel, que albergan la mayoría de PCs que integran un sistema cluster. Como se puede ver en la figura 1.4(der.), Intel ha lanzado cada dos años un nuevo procesador que ha doblado la potencia de su predecesor. Asimismo, como muestra la figura 1.5, prácticamente al mismo ritmo también se ha incrementado la relación tamaño/precio de la memoria RAM. En Junio del 2003 un chip de 512MB de SDRAM costaba igual que un chip de 256MB en Junio del 2002.

Un paso más en esta continua evolución está en los diferentes esfuerzos que se están realizando para integrar la memoria y el procesador en un único chip. Un ejemplo en esta línea de trabajo es el proyecto IRAM ("I*nteligent RAM*") [Pro02] o bien el procesador Alpha 21364 de Digital.

Este desarrollo justifica que si hace una década, la comunidad científica se planteó el uso de los clusters no dedicados para la ejecución de una única aplicación paralela (grado de multiprogramación paralelo igual a uno), hoy en día sea factible incrementar dicho grado de multiprogramación paralela (MPL). Este incremento del MPL comporta la necesidad de estudiar el rendimiento de las técnicas de coscheduling tradicionales en este nuevo escenario, así como proponer nuevos métodos, que permitan explotar eficientemente estos recursos de cómputo tanto para la ejecución de las aplicaciones paralelas como de las locales. Con este fin, frente a los esquemas de coscheduling tradicionales, donde se estudia únicamente la interacción entre el procesador y los eventos de comunicación, en este tesis se presenta una nueva propuesta de coscheduling donde las decisiones son tomadas en función de los eventos de

comunicación y procesador, del estado del subsistema de memoria, de la información de actividad del usuario local, así como de la información asociada a nodos remotos.

1.1.3. Sistema operativo

Dado nuestro interés en compartir los recursos de cómputo de un cluster entre las aplicaciones distribuidas y las aplicaciones asociadas a los usuarios locales de cada uno de los nodos, el sistema operativo (s.o.) de cada nodo debería cumplir los siguientes requerimientos mínimos:

- **Multiprogramado**. Esta característica presente en la mayoría de los s.o. actuales, es imprescindible para soportar la ejecución de múltiples procesos, ya sean locales o bien paralelos.
- **Multiusuario**. Dado que nuestro trabajo radica en que coexistan en una misma máquina tanto procesos pertenecientes al usuario local como al usuario de las aplicaciones distribuidas, el s.o. debe dar soporte para la gestión de múltiples usuarios.
- **Tiempo compartido**. Una de las prioridades de nuestro trabajo es garantizar un buen tiempo de respuesta al usuario local, normalmente caracterizado por sus necesidades de interacción. Con este fin, el s.o. subyacente debe disponer de esta característica.
- **Extensibilidad**. El s.o. debe permitir la integración de nuevos mecanismos de coscheduling que garanticen unos buenos tiempos de ejecución de las aplicaciones distribuidas. Ésto comportará la modificación de algunos de los subsistemas que lo integran, así como la creación de nuevas llamadas a sistema y estructuras de datos que incorporen la información necesaria para la gestión de los mismos.

El s.o. Linux cumple todos los requerimientos anteriores. A los anteriores requerimientos se han de añadir las razones esgrimidas a continuación, para justificar la elección del s.o. Linux como plataforma donde implementar nuestras propuestas:

1. **Código abierto**. Esta característica nos ha permitido incorporar todas aquellas modificaciones necesarias para permitir el coscheduling de las aplicaciones distribuidas.
2. **Gratuidad**. El uso del kernel de Linux es gratis, lo que obviamente repercute favorablemente en el costo de nuestra investigación.

3. **Documentación**. Hoy en día, el núcleo de Linux está ampliamente documentado lo que facilita, en gran medida, su estudio para la realización de las correspondientes modificaciones de acuerdo con nuestro fin.

4. **Investigación y desarrollo**. Linux es un s.o. en constante evolución, prueba de ello es el continuo lanzamiento de nuevas versiones que integran aquellas mejoras aportadas por la comunidad científica.

5. **Aceptación en la comunidad Cluster**. Hoy en día, Linux es el s.o. más utilizado en este tipo de entornos. Prueba de ello es el gran número de clusters gestionados por el s.o. Linux que aparecen en la lista de los 500 supercomputadores más rápidos del mundo [top02].

1.1.4. Cluster middelware y entornos de programación distribuida

El uso de un cluster para la ejecución de aplicaciones distribuidas comporta la instalación de una nueva capa, denominada "*middelware*", entre el s.o. y el nivel de usuario, que ofrezca un conjunto de servicios orientados al usuario de las aplicaciones distribuidas. De este modo, el cluster, desde el punto de vista del usuario de las aplicaciones distribuidas, puede ser tratado como una máquina virtual paralela. Entre estos servicios podríamos destacar la monitorización y administración de todos los recursos presentes a lo largo del cluster, la ejecución de aplicaciones paralelas, el mapping, checkpointing y scheduling de las aplicaciones distribuidas, etc...

Sobre la capa de *middleware* se ejecutan las aplicaciones distribuidas. Actualmente, la programación paralela de paso de mensajes es el modelo más ampliamente utilizado en entornos NOW. Bajo dicho modelo, cada programa paralelo consiste en un conjunto de procesos, escritos en un lenguaje secuencial, que cooperan entre si por medio de llamadas a funciones de envío y recepción de mensajes implementadas en una librería. Estos procesos que se intercambian mensajes entre si para la realización de un determinado trabajo serán denominados, a lo largo del texto, *procesos cooperantes*.

Actualmente, las dos librerías de paso de mensajes más utilizadas son el estándar de facto PVM ("*Parallel Virtual Machine*") [GBD+94], desarrollado e implementado en el Oak Ridge National Laboratory y el estándar MPI ("*Message Passing Interface*") [For02], definido por el fórum MPI. Una detallada descripción entre ambos entornos puede ser encontrada en [GKP96]. En la tabla 1.1 se resumen las principales diferencias entre ambos.

PVM	MPI
Ambas se ejecutan en MPPs y Clusters heterogéneos	
Concepto de máquina virtual	No posee dicha abstracción
Entorno de paso de mensajes	Especificación de paso de mensajes
Portabilidad sobre rendimiento	Rendimiento sobre flexibilidad
Topología no especificada	Topología lógica de interconexión
Tolerancia a fallos	Más susceptible a fallos
Solamente usa el protocolo TCP(UDP)/IP	Puede usar diferentes protocolos
Gestión de recursos, balanceo de carga y primitivas de control de procesos	Enfocado al intercambio de mensajes
Intercomunicación entre programas escritos en diferentes lenguajes	No soporta la comunicación entre diferentes lenguajes

Tabla 1.1: Comparación entre PVM y MPI.

Nuestro estudio no está dirigido a la mejora de ninguno de los aspectos listados en la tabla 1.1. De hecho, el estudio y la aplicabilidad de nuestras propuestas no depende de la elección tomada. Sin embargo, las facilidades de gestión de las tareas distribuidas y de los recursos de cómputo del cluster que incorpora PVM simplifica el control del cluster, así como la implementación de alguna de nuestra propuestas. En consecuencia, PVM ha sido el entorno de programación distribuida escogido.

1.1.5. Carga Local

La viabilidad en el uso de una red de PCs para la ejecución de aplicaciones paralelas depende, en gran medida, del número de recursos disponibles a lo largo de la misma. Con este fin, en la literatura científica hay presentes diferentes estudios respecto el grado de utilización tanto de los recursos de CPU [AES97] como de memoria [AS99].

La figura 1.6 muestra los porcentajes de utilización de CPU obtenidos en sendos estudios realizados por Acharya et al. [AES97] en dos clusters de 60 y 300 ws de la universidad de Berkeley y de la universidad de Wisconsin, respectivamente. Este estudio subdividía el tiempo de ocupación en tres porciones diferentes:

- **CPU**. Porcentaje de tiempo que la máquina presenta una carga media de CPU [3] superior a 0,3.

[3]Se entiende por carga media de CPU el número promedio de procesos en la cola de ejecución en el último minuto.

Figura 1.6: Porcentaje de utilización de CPU.

- **Interactiva (Inter)**. Porcentaje de tiempo con actividades de teclado y ratón, pero con carga de CPU inferior a 0,3.
- **Umbral de Reclutamiento (Reclut)**. Una máquina no se considera desocupada hasta que no pasa un determinado intervalo de tiempo, denominado *umbral de reclutamiento*, sin que no haya ni carga de CPU ni carga interactiva. El valor de dicho umbral es tomado diferente para cada una de las trazas obtenidas; mientras que en la universidad de Berkeley se considera un umbral de 3 minutos, en la universidad de Wisconsin, se toma un umbral de 15 minutos.

En ambas trazas destaca el bajo porcentaje de tiempo en el cual la CPU presenta una carga relativamente alta. Con respecto el porcentaje de desocupación, éste es superior en la traza de Berkeley, aunque si añadimos el porcentaje correspondiente al umbral de reclutamiento vemos que ambas trazas obtienen un tiempo similar de desocupación, alrededor del 60 %. Asimismo, es importante destacar el porcentaje de carga interactiva, alrededor del 25 %, lo que supone un tiempo donde la máquina sin estar libre dispone de un elevado porcentaje de recursos sin utilizar. Otros estudios realizados [LLM88] han obtenido similares resultados e incluso, porcentajes de desocupación superiores [Ryu01].

Los mismos autores realizaron un estudio similar respecto al porcentaje de utilización de la memoria [AS99]. Este estudio refleja el porcentaje de utilización del recurso de memoria respecto a una muestra compuesta de 52 ws correspondientes a dos clusters diferentes, con unos tamaños de memoria comprendidos entre los $32MB$ y los $512MB$. La memoria, en cada estación, es dividida en cuatro porciones diferentes:

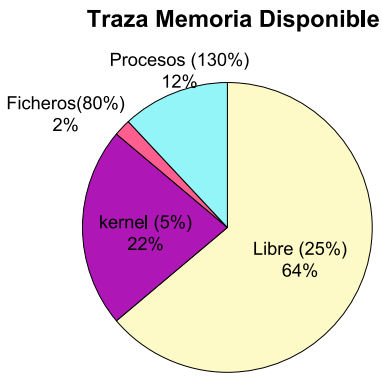

Figura 1.7: Porcentaje de utilización de la memoria.

1. *Kernel*, corresponde al porcentaje de memoria utilizado por el kernel.
2. *Ficheros*, corresponde al porcentaje utilizado por la cache de ficheros.
3. *Procesos*, corresponde al conjunto de memoria residente utilizado por los procesos activos en memoria.
4. *Libre*, corresponde al porcentaje de memoria no utilizada.

Cada una de las rebanadas del gráfico 1.7 muestra la media obtenida y la correspondiente desviación relativa (entre paréntesis). En primer lugar cabe destacar los elevados requerimientos de memoria por parte del s.o. En concreto, este valor está comprendido en el rango entre $10MB$ hasta los $49MB$ para las máquinas de $32MB$ y $256MB$, respectivamente. Por otro lado, destaca la baja utilización, en promedio, por parte de los procesos activos. Sin embargo, en este caso la desviación es superior al *100 %,* factor indicativo de la elevada variabilidad asociada con el tamaño de memoria disponible. Asimismo, el mismo estudio remarca que en contadas ocasiones y solamente durante pequeños intervalos de tiempo los usuarios consumen la práctica totalidad de la memoria disponible. Todo esto induce a los autores a la conclusión de que un elevado porcentaje de memoria está disponible durante una gran parte del tiempo.

En general, este elevado porcentaje de recursos de cómputo (CPU y memoria) no utilizados es debido a que las necesidades de los usuarios locales no han evolucionado al ritmo marcado por la tecnología. En esta línea

cabe destacar los estudios realizados por Wiley and Rosen [WR97] sobre el comportamiento de los usuarios de entornos ofimáticos ante los constantes cambios que experimentan los entornos en los que trabajan. El resultado de estos estudios refleja el alto grado de reticencia que presentan los usuarios de un sistema informático a cambiar los programas que utilizan. Esta conducta, conocida con el nombre de *tecnofobia*, también se refleja en los diferentes estudios realizados sobre el uso de los recursos de computación empleados por los usuarios locales. Comparando los trabajos anteriormente descritos, de Acharya et al. [AES97] que datan del año 1997, con el reciente trabajo de Ryu [Ryu01], realizado en el 2001, se observa como en el intervalo de cuatro años, transcurridos entre ambos estudios, los requerimientos de CPU prácticamente no se han alterado mientras que los de memoria han aumentado aproximadamente un 50 %, es decir muy por debajo del incremento de tamaño de las memorias marcado por el propio mercado tecnológico.

De los anteriores estudios se desprende que un elevado porcentaje de recursos de cómputo (CPU y memoria) no son utilizados por los usuarios locales y, por tanto, están disponibles para la ejecución de las aplicaciones paralelas. Sin embargo, la elevada variabilidad de estos recursos disponibles muestran la necesidad de implementar mecanismos que permitan una asignación dinámica de los recursos disponibles a lo largo del cluster para cada una de las aplicaciones paralelas en ejecución.

1.2. Técnicas de Planificación de Aplicaciones Paralelas en un Cluster no Dedicado

El binomio formado por la potencia de cómputo asociada a los entornos cluster actuales, junto con la baja tasa de utilización de los mismos por parte de los usuarios locales, han incentivado a la comunidad científica a estudiar la posibilidad de ejecutar aplicaciones paralelas en un cluster no dedicado. La convivencia de ambos tipos de aplicaciones en un mismo cluster ha comportado una revisión de las técnicas clásicas de planificación de procesos en entornos paralelos dedicados. En esta sección se describen las principales técnicas de planificación orientadas a sistemas multiprocesador/multicomputador en general y entornos cluster no dedicados, en particular, propuestas en la literatura científica.

En un entorno monoprocesador, la *planificación a corto plazo* es la actividad de decidir que proceso toma el control de la CPU. En cambio, en un sistema multicomputador, como es el caso de los clusters, el objetivo de la planificación, no solamente es decidir *cuándo* un proceso se ejecutará (*plani-*

ficación en el tiempo), si no que también deberá decidir *donde* será ejecutado dicho proceso (*planificación en el espacio*).

En un sistema paralelo multiprogramado, como el que nos ocupa, donde varias aplicaciones distribuidas se ejecutan concurrentemente, los recursos de cómputo pueden ser compartidos, tanto en el *espacio* como en el *tiempo*. Teniendo en cuenta este hecho, Feitelson [Fei97] clasificó las técnicas de planificación en un sistema paralelo, en función del modo en que los recursos son compartidos: tiempo compartido, espacio compartido o bien una combinación de ambos. La tabla 1.2 muestra una versión reducida de dicha clasificación, con un ejemplo de una máquina real para cada uno de los casos particulares. En primer lugar, cabe destacar el hecho de que los mecanismos de tiempo compartido son totalmente independientes de los de espacio compartido. De este modo, las técnicas de planificación se pueden clasificar mediante una matriz bidimensional, donde en las columnas se muestran las técnicas de tiempo compartido y en las filas las técnicas de espacio compartido. El hecho de que prácticamente todas las celdas de la matriz estén ocupadas, con ejemplos de máquinas reales, ratifica la independencia de ambos esquemas de planificación.

	tiempo compartido			
	si			no
espacio	PEs independientes			
compartido	cola global	cola local	coscheduling	
si	Mach	Paragon, Meiko	Medusa, SGI	IBM SP2,
no	IRIX on SGI	StarOS, Payche	MasPar MP2	Illiac IV

Tabla 1.2: Clasificación de las técnicas de planificación.

Las técnicas de espacio compartido dividen la máquina en un conjunto de particiones de manera que, en cada una de ellas, se ejecutan una o varias aplicaciones. Son numerosas las contribuciones realizadas en este campo [FST94, FR96, TWY92], sobre todo en lo que se refiere al estudio del tamaño óptimo de las particiones. Teniendo en cuenta dichos trabajos, estas particiones pueden definirse siguiendo los siguientes modelos:

- **Fijas**. El tamaño de la partición es definida por el administrador del sistema y solamente puede ser modificado reiniciando la máquina.

- **Variables**. El tamaño de la partición es determinado por el propio usuario en el instante en que la aplicación es lanzada.

- **Adaptables**. El tamaño de la partición es determinado por el planificador en el instante de ejecución del trabajo, de acuerdo con la carga del sistema y teniendo en cuenta las peticiones del usuario.

- **Dinámicas**. El tamaño de la partición puede cambiar durante la ejecución de un trabajo, de acuerdo con los requerimientos del propio trabajo y de la carga del sistema.

Esta tesis aborda el problema de la planificación de trabajos en el tiempo, una vez los trabajos paralelos han sido divididos en un conjunto de procesos y éstos han sido asignados a un conjunto de procesadores. De este modo, las técnicas desarrolladas a lo largo de esta tesis asumirán cualquier tamaño y tipo de partición del cluster.

Centrándonos en las técnicas de tiempo compartido mostradas en la tabla 1.2, Feitelson las divide en dos grandes grupos: las que tratan cada elemento de procesamiento independientemente y las que toman las decisiones de una manera global a lo largo de un conjunto de procesadores (*coscheduling*). A su vez, los mecanismos de planificación independientes se subdividen entre aquellos que trabajan con colas locales, propios de máquinas de memoria distribuida, y aquellos que usan una cola global compartida por todos los PEs, propios de máquinas de memoria compartida.

Teniendo en cuenta que nuestro trabajo se desarrolla en entornos cluster, en las próximas secciones nos centraremos en aquellas técnicas de planificación propias de estos entornos: PEs independientes con colas locales y coscheduling.

1.2.1. Técnicas de planificación basadas en PEs independientes con colas locales

En un entorno cluster no dedicado, un aspecto a tener en cuenta es la reticencia del usuario local de cada workstation para que su máquina, normalmente utilizada en exclusividad, pase a ser compartida por diferentes usuarios. Este hecho comporta que garantizar el rendimiento de las aplicaciones del usuario local sea un objetivo primordial de todas aquellas técnicas que explotan la potencia de cómputo remanente en un cluster no dedicado para la ejecución de aplicaciones paralelas. Las investigaciones realizadas con este fin se enmarcan en dos líneas diferentes: aquellas basadas en el uso de *técnicas de preempción local* o bien aquellas que utilizan *técnicas de preempción con migración de procesos*.

Las técnicas basadas en migración de procesos se basan en suspender la ejecución de las tareas remotas, en aquellos nodos ocupados por el usuario

local, y reanudar posteriormente dichas tareas en otros nodos libres. Por tanto, la migración de procesos comporta la elección del mejor nodo disponible a donde migrar los procesos. Esta decisión conlleva que las técnicas basadas en la migración deben mantener información de los nodos desocupados y, por tanto, susceptibles de recibir un proceso migrado. La cuestión es cuanto tiempo (conocido como umbral de reclutamiento) un nodo debe de estar libre para ser considerado como nodo desocupado. Intuitivamente, con objeto de maximizar el rendimiento de las aplicaciones distribuidas, la mejor opción sería tomar un umbral igual a cero. Sin embargo, Douglis y Ousterhout [DO91] demostraron que en máquinas que han sido recientemente utilizadas por el usuario local, existe una elevada probabilidad de que el usuario local retorne y por tanto, la máquina deje de estar disponible para la migración. A partir de diversos estudios realizados [DO91, ML91], se desprende que un umbral de reclutamiento mínimo de $180sg$ es necesario para considerar que una máquina está desocupada y por tanto puede recibir procesos migrados. Arpaci et al. [ADV+95] estudiaron el elevado coste que supone el proceso de migración; por ejemplo, *Condor* [LLM88] requiere aproximadamente dos minutos [LS92]. Este elevado coste induce al sistema *Condor* a tomar un umbral de reclutamiento de 15 minutos. Con objeto de reducir el coste de migración, algunos entornos han optado por implementar la migración en el espacio del kernel; un ejemplo de esta alternativa sería *el sistema Sprite* [OCD+88]. Un problema asociado a los entornos citados anteriormente, es que están limitados a entornos homogéneos. En este sentido, *V-System* [TLC85] y *Stardust* [CP97] permiten migración entre máquinas heterogéneas, a costa de aumentar considerablemente el tiempo de migración y de aumentar la complejidad del compilador[4].

Asociado a la migración de procesos, están las técnicas de balanceo de carga ("*load balancing*"). El propio comportamiento tanto de los usuarios locales como de las carga distribuidas que se ejecutan en el cluster, determinan que un cluster sea un entorno caracterizado por una alta variabilidad, lo que comporta que haya una alta probabilidad de que algunos nodos estén muy cargados, mientras que otros nodos estén prácticamente ociosos. Por este motivo, técnicas basadas en migrar procesos con objeto de balancear la carga a lo largo del cluster deberían suponer una aumento en el rendimiento del cluster. Un algoritmo de *load balancing* consiste básicamente de tres fases:

1. *Captura de información* necesaria para tomar las decisiones sobre qué procesos migrar y donde deben ser migrados dichos procesos, junto con

[4]Debemos tener en cuenta que permitir la migración entre máquinas heterogéneas supone, entre otras cosa, que el compilador debe retener información sobre que estructuras contienen punteros y a donde apunta cada puntero.

la estrategia utilizada para distribuir dicha información a lo largo del cluster.

2. *Elección del proceso* a migrar de acuerdo con los niveles de carga de los diferentes nodos. Por ejemplo, el proceso escogido debería tener pendiente suficiente computación para compensar el overhead producido por la migración.

3. *Selección del nodo* a donde migrar el proceso escogido en el paso anterior.

El entorno *MOSIX* [BL98] es un ejemplo de sistema orientado al balanceo de carga. MOSIX está basado en el kernel de UNIX, al cual se le han añadido nuevas funcionalidades como son la migración de procesos y mecanismos de balanceo de carga. Con objeto de aumentar su escalabilidad, el sistema es totalmente distribuido. Al contrario que los entornos citados anteriormente, MOSIX no distingue entre los usuarios locales y los usuarios distribuidos, de manera que todos los procesos (tanto locales como distribuidos) disfrutan del mismo nivel de servicios. Por este motivo, la filosofía utilizada por MOSIX se aleja de nuestros propósitos. Cabe destacar, que las teć nicas de balanceo de carga han sido ampliamente estudiadas en la literatura; entre estos sistemas destacamos por su amplia difusión, el sistema *NOW* [ACP+95] y el Sistema *Dynamic PVM* [OSHH96].

Con objeto de reducir el overhead implícito en el proceso de migración de las tareas remotas, diversos autores [MS70, KC91] han apostado por el uso de las técnicas de preempción local. Éstas se caracterizan por priorizar la asignación de recursos locales a las tareas del usuario local frente a las remotas o paralelas; solución propuesta por *Stealth Distributed Scheduler* [KC91]. El objetivo de Stealth es ir cediendo a los procesos distribuidos todos aquellos recursos que no sean utilizados por los procesos locales. De este modo, cuando el usuario local reclama el uso de su máquina, el trabajo remoto continua ejecutándose con baja prioridad. En concreto, Stealth prioriza el uso de la CPU, memoria principal y la cache de ficheros asociadas el sistema operativo *MACH*. Sin embargo, diferentes estudios [CDD+94, FR92] demuestran, que en este tipo de entornos, las aplicaciones distribuidas con elevadas comunicaciones sufren excesivos retardos debido a la ausencia de coplanificación entre los procesos cooperantes que se comunican o sincronizan entre si. Esto es debido a que los mensajes recibidos para un proceso no planificado son almacenados en un buffer temporal, de manera que la correspondiente respuesta no se realiza hasta que dicho proceso vuelve a ser planificado; lo que comporta que el proceso emisor del mensaje quede bloqueado en espera de la respuesta correspondiente.

Una contribución interesante en esta línea, ha sido el trabajo realizado por Ryu et al. en *Linger_ Longer* [RH98], donde se intentan conjugar las ventajas que aportan tanto las técnicas de preempción local como el uso de la migración. En Linger_longer, una vez el usuario reanuda su trabajo, las tareas distribuidas continuarán ejecutándose con muy baja prioridad, es decir, a semejanza del sistema Stealth. Sin embargo, Linger_Longer permite la migración de procesos solamente en el caso de que los beneficios aportados por la migración superen su coste. Los resultados obtenidos en simulación mostraron que las estrategias basadas exclusivamente en preempción local superan ampliamente a las estrategias basadas en migración, siempre y cuando la tasa de utilización del procesador por parte del usuario local sea menor que el 30 %.

1.2.2. Técnicas de coscheduling

En sistemas paralelos multiprogramados, los procesos pertenecientes a una misma aplicación cooperan entre si, mientras compiten por los recursos compartidos con los procesos pertenecientes a aplicaciones rivales. En este contexto, la cuestión planteada por diferentes investigadores es si el sistema operativo debería ser capaz de distinguir entre los procesos cooperantes y rivales y actuar en consecuencia. Los mecanismos de planificación descritos en esta sección reflejan la creencia de que es importante planificar juntos los procesos cooperantes; opinión que es compartida por un gran número de fabricantes e investigadores [LAD+96, Ste92, HTI+96, FR92, Sol02, AD01, SW95].

John Ousterhout estudió la analogía existente entre la gestión de memoria en un entorno monoprocesador multiprogramado y la gestión de procesadores en un entorno multiprocesador multiprogramado [Ous82]. Esta analogía se basa en la observación de que las aplicaciones paralelas requieren que un conjunto de procesos se ejecuten simultáneamente, exactamente como una aplicación local en un entorno monoprocesador requiere que todas las páginas que forman su *working set* estén residentes simultáneamente en memoria [Den68]. En dicho estudio, Ousterhout comprobó que si una aplicación paralela no recibe suficientes procesadores para planificar cada tarea en un PE distinto y a su vez, estas tareas no son planificadas simultáneamente, el rendimiento de dicha aplicación se verá seriamente afectado debido a los requerimientos de comunicación y sincronización entre sus procesos cooperantes. Este comportamiento es debido a que un proceso en ejecución puede quedar bloqueado esperando las respuestas de aquellos procesos cooperantes que no están simultáneamente planificados en los nodos remotos; con la consiguiente perdida de tiempo asociada a la realización de múltiples cambios de contexto

innecesarios. A raíz de este trabajo, Ousterhout introdujo el concepto de *coscheduling de aplicaciones paralelas*. De acuerdo con la definición de coscheduling de Ousterhout, una aplicación está coplanificada ("*coscheduled*") si todos los procesos que la integran están ejecutándose simultáneamente en diferentes procesadores.

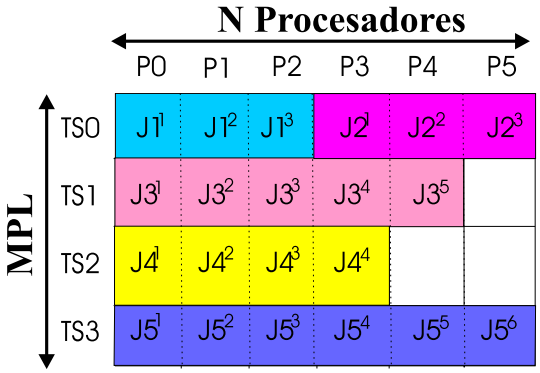

Figure 1.8: Ejemplo de matriz de Ousterhout.

Ousterhout propuso un algoritmo de coscheduling conocido como *algoritmo de la Matriz de Ousterhout*. En este algoritmo se define una matriz formada por un eje temporal de MPL filas y un eje espacial de N columnas, donde MPL es el grado de multiprogramación y N el número de procesadores disponibles en el sistema. En la matriz de Ousterhout, cada columna define los procesos (Ji^k) asignados a un procesador dado (Pi), mientras que cada fila define tanto los trabajos (Ji) que serán ejecutados durante un quantum de tiempo determinado (TSi), como el mapping de los procesos que integran dichos trabajos a los correspondientes procesadores. A modo de ejemplo, la figura 1.8 muestra una matriz con seis procesadores ($N = 6$) y un grado de multiprogramación de 4 ($MPL = 4$). De esta manera, cada vez que se ha de asignar un nuevo trabajo (Ji) al sistema se busca, comenzando por la fila cero, una fila con igual número de celdas libres que procesos tenga el trabajo a ser asignado. Una vez asignado dicho trabajo, el algoritmo utiliza un mecanismo de *round-robin* para planificar las diferentes filas de la matriz. De este modo, teniendo en cuenta el ejemplo de la figura 1.8, durante el quantum de tiempo *TS0* se ejecutarán los procesos pertenecientes a los trabajos J1 y J2, de manera que al finalizar dicho quantum se producirá un cambio de contexto global, de modo que el trabajo J3 será planificado en los procesadores *P0, P1, P2, P3 y P4* durante el siguiente quantum *TS1*, y así sucesivamente.

Numerosos autores han utilizado el término *Gang scheduling* para referirse

a la necesidad de coscheduling de las aplicaciones paralelas. Feitelson [Fei97] define Gang scheduling como un esquema de planificación que combina las tres características siguientes:

1. Las tareas son agrupadas en grupos, considerando en la mayoría de los casos una aplicación distribuida como un único grupo.

2. Las tareas dentro de cada grupo son ejecutadas simultáneamente en distintos PEs, asignando una tarea de cada grupo a un PE distinto.

3. La técnica de tiempo compartido es utilizada para compartir cada PE entre los distintos grupos, de manera que todas las tareas dentro de un grupo son planificadas simultáneamente a lo largo de la máquina paralela.

Acorde con esta definición, se desprende que el término de gang scheduling se puede considerar como un sinónimo del concepto de coscheduling definido por Ousterhout. De este modo, a lo largo de esta tesis ambos términos serán tratados como sinónimos.

Una segundo modo de entender el Coscheduling, más flexible que la anterior propuesta de Ousterhout, fue propuesto por Sobalvarro [SW95] como sigue: *Coscheduling* asegura que ningún proceso esperará a un proceso cooperante debido a eventos de comunicación y sincronización. La principal diferencia respecto a la definición dada por Ousterhout radica en el hecho de que no todos los procesos que integran una aplicación deben estar coplanificados, si no que solamente se deben coplanificar aquellos procesos que estén cooperando en un determinado instante de tiempo.

De acuerdo con las dos definiciones anteriores, muchas han sido las contribuciones aportadas en la literatura con el objetivo de implementar el coscheduling de aplicaciones paralelas en entornos cluster, tanto dedicados como no dedicados. En la próxima sección, con objeto de enmarcar nuestro trabajo, se describirán las principales aportaciones realizadas hasta la fecha.

1.3. Estado del Arte de las Técnicas de Coscheduling

La tabla 1.3 muestra la taxonomía de las técnicas de coscheduling más relevantes presentadas en la literatura científica. En el eje de abcisas, dichas técnicas se clasifican atendiendo a la metodología utilizada por las mismas para localizar a los procesos cooperantes y controlar que éstos se ejecuten

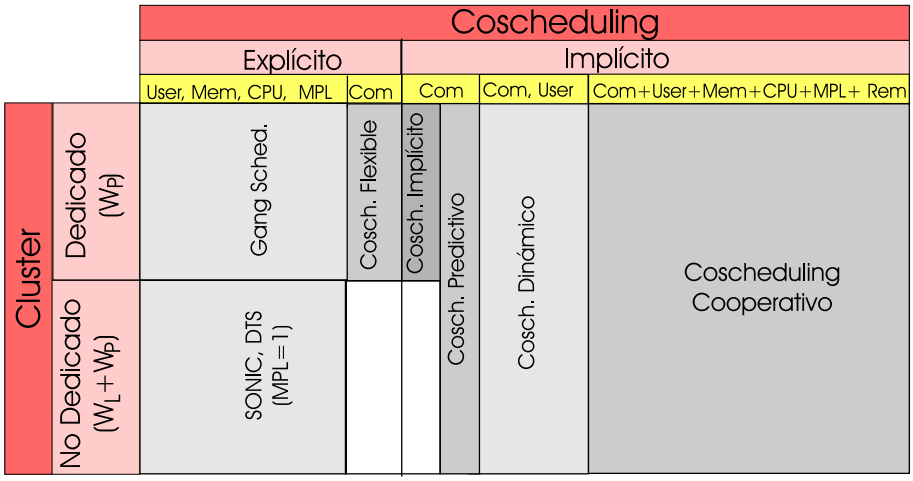

Tabla 1.3: Taxonomía de las técnicas de coscheduling. W_p : Carga paralela, W_L : Carga local, *Com*: Comunicación, *User* : Actividad del usuario local, Mem: Memoria, CPU: Longitud del quantum, *MPL*: Grado de multiprogramación paralela, *Rem*: Información de sistemas remotos.

simultáneamente [PKB00]; mientras que en el eje de ordenadas se agrupan en función de su usabilidad en un entorno cluster dedicado/no dedicado.

En función de la naturaleza del mecanismo de control utilizado para establecer la coplanificación entre procesos cooperantes, las técnicas de coscheduling se clasifican en dos grupos:

- **Técnicas de coscheduling con control explícito.** Acorde con la definición de Ousterhout, dada en la sección anterior, la implementación del coscheduling requiere que un cambio de contexto global se realice simultáneamente a lo largo de toda la máquina paralela; de esta manera, cada aplicación distribuida tiene la sensación de que se está ejecutando en un entorno dedicado durante la correspondiente rebanada de tiempo. En función de como se realicen dichos cambios de contexto se pueden distinguir dos maneras diferentes de coordinación, una *estática* y otra *dinámica*. En la alternativa estática, los diferentes planificadores locales sincronizan sus relojes de acuerdo con un nodo master, de manera que todos los nodos realizan los correspondientes cambios de contexto en el mismo instante de tiempo y de una manera prefijada por el nodo master. En la alternativa dinámica, el nodo master marca el correspondiente cambio de contexto mediante el envío, en *broadcast,* de una señal de control a todos los nodos.

- **Técnicas de coscheduling con control implícito.** Estas técnicas se ajustan a la definición de coscheduling propuesta por Sobalvarro. Las decisiones de planificación son tomadas por cada planificador local en tiempo de ejecución, de acuerdo con la ocurrencia de ciertos eventos, tanto locales, como pueden ser: eventos de comunicación (*Com*), de memoria (*Mem*), de CPU (*CPU)*, de actividad de usuarios locales (*User*) o grado de multiprogramación (*MPL*), así como eventos ocurridos en nodos remotos (*Rem*). Cada nodo, a partir del análisis de dichos eventos realiza una predicción acerca de que procesos se están ejecutando en los nodos remotos con el objetivo de intentar detectar el conjunto de procesos cooperantes y asignar los recursos en consecuencia.

La elección entre ambas alternativas no es ni mucho menos obvia. Mientras que las técnicas con control explícito aseguran la coplanificación de los procesos cooperantes, son difíciles de ser adaptadas a entornos tan dinámicos como puede ser un cluster no dedicado. Por otro lado, las técnicas con control implícito son mucho más flexibles, aspecto que permite que sean implementadas más fácilmente en un cluster no dedicado; sin embargo, atendiendo a que sus decisiones de planificación son basadas a partir de la realización de predicciones, no aseguran el coscheduling de las tareas cooperantes. En definitiva, la elección entre una u otra opción dependerá de los requerimientos del sistema a implementar.

En las siguientes secciones se describirán las principales aportaciones mostradas en la tabla 1.3, atendiendo al mecanismo de control que utilizan y haciendo especial énfasis en aquellas técnicas orientadas a entornos cluster no dedicados.

1.3.1. Coscheduling con control explícito

Hasta la fecha, la técnica de gang scheduling ha sido ampliamente utilizada en entornos multiprocesador/multicomputador comerciales. Entre los sistemas que la implementan, podríamos citar, el Connection Machine CM-5 [Cor92], el Intel Paragon [Div94], el Meiko CS-2 [SS95] y el Cray T3E [LG97].

Asimismo, son numerosos los estudios realizados sobre el rendimiento de las técnicas de gang scheduling en entornos cluster dedicados. Petrini et al. [FPCF01] implementaron un sistema de planificación de gang scheduling sobre un cluster con una red de interconexión Quadrics (QsNET) [PHFG01]. Sobre esta plataforma estudiaron el nivel de sensibilidad de las técnicas de gang scheduling con respecto al nivel de multiprogramación (MPL), los requerimientos de memoria y la longitud del quantum. En sus conclusiones, se remarca la estabilidad que presenta gang scheduling a las variaciones de

los requerimientos de memoria de las aplicaciones, siempre y cuando no se sobrepase el tamaño de la memoria física, pero en cambio se destaca su alta variabilidad con respecto a la longitud del quantum. Un estudio similar centrado sobre los requerimientos de memoria en un entorno gang scheduling fue realizado por Batat y Feitelson en [BF00]. En dicho estudio, se remarca la necesidad de agrupar los procesos acorde a sus requerimientos de memoria.

Asimismo caben destacar los numerosos trabajos realizados en el estudio de la fragmentación en entornos cluster [Fei96, FFPF03, Aid00]. La fragmentación es un problema intrínseco con la técnica de gang scheduling, que cobra especial interés en los entornos cluster debido a la naturaleza heterogénea de los mismos. La fragmentación ocurre cuando diferentes PEs están ociosos en una misma rebanada de tiempo pero éstos no son suficientes para poder albergar ningún trabajo que esté en espera de ser ejecutado. En esta línea, destacamos el reciente trabajo realizado por Fractenberg et al. [FFPF03], donde se presenta una variación de la técnica de gang scheduling denominada "*Flexible Coscheduling*". La principal novedad aportada por esta técnica radica en que los trabajos son clasificados, en tiempo de ejecución, de acuerdo con sus requerimientos de comunicación en dos grupos diferentes: aquellos que necesitan coscheduling y aquellos que no tienen ninguna necesidad del mismo. De acuerdo con esta clasificación, solamente se aplica el gang scheduling para aquellos procesos que lo necesitan, de manera que los procesos sin necesidad de coscheduling podrán ser ejecutados en cualquier rebanada de tiempo. De este modo se consigue reducir la fragmentación y aumentar el rendimiento global del cluster. Aunque hasta la fecha solamente ha sido probado en un entorno cluster dedicado, los mismos autores mencionan la posibilidad de utilizar dicho sistema en un entorno cluster no dedicado.

El uso de la técnica de gang scheduling comporta que las diferentes aplicaciones a ejecutar sean conocidas de antemano, cosa difícil de conocer en un entorno tan variable como es un cluster no dedicado. Por esta razón, los esfuerzos realizados en entornos cluster no dedicados han estado orientados hacia la idea de crear una máquina virtual paralela (MVP) con una doble funcionalidad: ejecutar las tareas paralelas con un bajo overhead y un buen tiempo de respuesta de las tareas locales. En esta área de investigación se centran los trabajos *SONIC* y *DTS*, descritos en [Pol96, SGHL00], respectivamente. Estos entornos se caracterizan porque todos los nodos asignan un mismo porcentaje de recursos a las tareas locales y el resto a las distribuidas; siendo fijado este porcentaje explícitamente por un nodo master. Ambos entornos solamente permiten ejecutar una única aplicación distribuida ($MPL = 1$) junto con la carga local. Esta restricción en el grado de multiprogramación paralela es debida a la extrema rigidez que presentan estos esquemas; de modo que si se aumentase el grado de multiprogramación, el

usuario local podría padecer importantes retardos. Asimismo, estos esquemas no son capaces de aprovechar todos los recursos de cómputo no utilizados por los usuarios locales, de manera que frecuentemente ocurre que el usuario local no utiliza su porcentaje de tiempo asignado y, en consecuencia, la máquina queda totalmente ociosa. Un segundo problema asociado con este tipo de entornos, es la escalabilidad de los mismos debido al propio mecanismo de sincronización utilizado. Finalmente, un tercer problema a destacar es el elevado overhead que supone realizar un cambio de contexto global en máquinas con un elevado número de nodos [MCF+98], hecho que comporta que estos sistemas tengan que trabajar con unas rebanadas de tiempo del orden de segundos; longitudes de quantum impensables en entornos que requieren elevada interactividad, como es el caso de los entornos NOW no dedicados. Asimismo, este overhead debido al cambio de contexto puede incrementarse en el caso de que los nodos no dispongan de suficiente memoria para encajar todas las aplicaciones activas.

Por todos estos motivos, los trabajos de coscheduling más recientemente desarrollados en entornos cluster no dedicados se han centrado en el uso de técnicas con control implícito.

1.3.2. Coscheduling con control implícito

La tabla 1.3 ilustra las técnicas de coscheduling con control implícito, más significativas, presentadas en la literatura. En esta sección se describirán todas aquellas técnicas de coscheduling basadas, principalmente, en la ocurrencia de eventos de comunicación, las cuales serán denominadas, a lo largo del presente texto, como técnicas de *coscheduling tradicionales*. A partir del análisis de las restricciones asociadas a dichas técnicas se justificará la necesidad de la nueva propuesta de coscheduling, denominada *CoScheduling Cooperativo*, que tenga en cuenta el estudio de otros eventos.

En [SW95], Sobalvarro demostró mediante el uso de herramientas de simulación que era posible alcanzar el coscheduling de procesos que se comunican entre si a partir de la información obtenida mediante la monitorización y análisis de los eventos locales ocurridos en cada nodo. De este modo, se eliminaba la necesidad de intercambiar información y señales de control entre los nodos, propia de los algoritmos de coscheduling con control explícito. A partir de este trabajo, Sobalvarro introdujo una nueva técnica de coscheduling denominada *coscheduling bajo demanda* ("*demand-based coscheduling*"). Bajo este esquema, los procesos cooperantes son coplanificados solamente cuando se comunican entre si; de esta manera, los eventos de comunicación marcan la necesidad de planificación de un proceso. Sobalvarro distinguió dos posibles métodos de coscheduling bajo demanda, *coscheduling Dinámico* y *cosche-*

duling Predictivo.

El *coscheduling Dinámico* utiliza la llegada de un mensaje como una señal para planificar el proceso destinatario del mismo, incluso causando la preempción del proceso actualmente en ejecución. La lógica detrás de este concepto consiste en que cuando un proceso recibe un mensaje es debido a que el proceso emisor está planificado en un nodo remoto. Sobalvarro mediante simulación probó dos posibles implementaciones del coscheduling Dinámico: en la primera de ellas, denominada "*always-schedule dynamic coscheduling*", la recepción de un mensaje siempre causa un cambio de contexto a favor de la tarea destino del mismo; mientras que en la segunda implementación, denominada "*equalizing dynamic coscheduling*" se limita el número de veces que una tarea paralela puede adelantar a una local; de esta manera se evita una posible inanición de las tareas locales con respecto a las tareas paralelas con elevada frecuencia de comunicación.

La estrategia de "*equalizing dynamic coscheduling*" fue evaluada por Sobalvarro en un entorno cluster no dedicado formado por siete estaciones de trabajo (SPARCstation-2) conectadas por una red Myrinet y con el protocolo de comunicación *Active Message*s [SPWC98]. En esta implementación se modificó el *firmware* de la tarjeta de red con objeto de implementar el coscheduling Dinámico. De este modo, cada vez que la tarjeta recibe un mensaje para un proceso distinto al que se está ejecutando invoca, mediante la llamada a una interrupción, la ejecución del gestor de comunicaciones con objeto de aumentar la prioridad del proceso receptor del mensaje. Una importante limitación del sistema utilizado por Sobalvarro es su total dependencia con el hardware de comunicación empleado. Una segunda implementación de esta técnica fue realizada por Solsona et al. [Sol02]. En dicho trabajo, el coscheduling Dinámico fue implementado en el propio kernel del s.o. Linux. En esta implementación, cada vez que un proceso recibe un mensaje, su prioridad es incrementada por el propio planificador de manera que dicho proceso sea planificado inmediatamente. Asimismo, con objeto de proteger el rendimiento de las tareas locales, se limita el máximo número de adelantos que pueden sufrir las tareas locales en espera a ser planificadas. En un cluster de 4 máquinas conectadas por una red Fast Ethernet, Solsona compara el rendimiento de la técnica de coscheduling Dinámico con respecto al Linux original, reduciendo, en la mayoría de casos, el tiempo de ejecución de las aplicaciones paralelas por debajo de la mitad y conservando el tiempo de respuesta de las aplicaciones locales, dentro de unos límites aceptables por el mismo.

Con objeto de permitir la portabilidad del sistema, otras implementaciones del coscheduling Dinámico [GRV01, YJ01] han sido realizadas en el propio espacio del usuario, en concreto en la librería de paso de mensajes

PVM o bien MPI. Sin embargo, los resultados alcanzados en ambas implementaciones no son tan brillantes como los obtenidos en las implementaciones descritas anteriormente; de manera que en ambos casos se obtienen unas ganancias comprendidas en un intervalo entre el 20 % y 30 % con respecto a la ejecución no coordinada (Linux+PVM/MPI original).

Finalmente, en entornos gestionados por la técnica de coscheduling Dinámico cabe destacar el bajo slowdown asociado a los trabajos locales debido a la ejecución de las tareas paralelas. Este comportamiento se refleja en los resultados obtenidos en tres [SPWC98, Sol02, GRV01] de las cuatro implementaciones descritas anteriormente. El slowdown obtenido por las tareas locales cuando son ejecutadas junto con una aplicación distribuida ($MPL = 1$) es, en los trabajos mencionados, menor que 1,5. Asimismo, estos mismos trabajos evidencian que un incremento en el grado de multiprogramación paralela en un entorno cluster no dedicado supone tener en cuenta los requerimientos de memoria de las aplicaciones, tanto locales como distribuidas, con el objetivo de no desbordar la memoria principal y como consecuencia perjudicar, en exceso, el trabajo de los usuarios locales.

La segunda técnica de coscheduling bajo demanda definida por Sobalvarro, denominada *coscheduling Predictivo*, se basa en predecir qué procesos es probable que se comuniquen entre si. Recordando el símil realizado por Ousterhout con el subsistema de memoria virtual, el coscheduling Predictivo, al igual que el algoritmo LRU en un sistema de paginación bajo demanda, trata de predecir el comportamiento futuro a partir del comportamiento pasado. En nuestro conocimiento, el trabajo de Solsona et al. en [SGHL01c] es la única implementación de coscheduling Predictivo realizada hasta la fecha. En esta implementación, la necesidad de coscheduling de cada tarea se predice a partir de su frecuencia de comunicación pasada. Los resultados obtenidos por Solsona demostraron el potencial de dicha técnica en entornos dedicados, obteniendo mejores prestaciones que el coscheduling Dinámico a medida que se aumentaba el grado de multiprogramación [Sol02]. Sin embargo, un problema intrínseco a la utilización de dicha técnica en entornos no dedicados es la elevada intrusión introducida en la ejecución de las tareas locales cuando el número de trabajos paralelos es incrementado.

Una variación del coscheduling Dinámico, llamada coscheduling Implícito ("*implicit coscheduling*"), es presentada por Arpaci-Dusseau en [ADCM98, ADC97, ADV+95]. En esta técnica, la tarea que ha enviado un mensaje realiza una espera activa (*spin*) durante un cierto intervalo de tiempo, de manera que si la respuesta es recibida antes de que el intervalo expire, dicha tarea continuará ejecutándose mientras, que en caso contrario, el proceso se bloqueará y otra tarea será planificada. Las diferentes implementaciones realizadas de dicha técnica [AD01, CADG+93, WADC99, SFHL00] han mostrado su via-

bilidad en entornos dedicados. Sin embargo, la espera activa que realiza cada tarea que envía un mensaje puede provocar una penalización excesiva en la ejecución de las tareas locales; hecho que desaconseja el uso de esta técnica en entornos no dedicados.

En [Ang00], Anglano compara por medio de la simulación ocho estrategias diferentes de coscheduling con control implícito, combinando técnicas de coscheduling Implícito y coscheduling Dinámico. En concreto, en función del comportamiento del proceso en espera de la recepción de un mensaje, se definen las siguientes cuatro estrategias:

- **Spin (S)**. El proceso realiza una espera activa indefinida hasta que recibe un mensaje.
- **Block (B)**. El proceso se bloquea y es despertado por el propio sistema operativo cuando llega el mensaje.
- **Spin-Block (IM)**. Esta estrategia se corresponde con el coscheduling Implícito puro explicado anteriormente.
- **Spin-Yield (SY)**. El proceso, como en el caso del spin-block, realiza una espera activa, pero una vez finalizada la misma no se bloquea, si no que simplemente disminuye su prioridad con el objetivo de que pueda ser planificado otro proceso distribuido preparado para ejecución.

Cada una de las cuatro técnicas anteriormente descritas, fue evaluada junto con dos estrategias diferentes ante la recepción de un mensaje por parte de una tarea: (1) no haciendo nada ante la recepción de un mensaje (*Loc*) y (2) incrementando, inmediatamente tras la recepción de un mensaje, la prioridad de planificación del proceso receptor (se corresponde al coscheduling Dinámico (*D*yn)). La tabla 1.4 sintetiza las diferentes combinaciones realizadas.

	¿Cómo esperar un mensaje?			
¿Qué hacer ante la llegada de un mensaje?	*Spin*	*Block*	*Spin-Block*	*Spin-Yield*
Loc	Loc-S	Loc-B	Loc-IM	Loc-SY
Dyn	Dyn-S	Dyn-B	Dyn-IM	Dyn-SY

Tabla 1.4: Estrategias en la espera de un mensaje.

Los resultados obtenidos por Anglano revelan que tanto en un entorno dedicado multiprogramado como en un entorno no dedicado con un grado de multiprogramación igual a uno, la mejor estrategia de espera es la del bloqueo

inmediato (B), mientras que los peores resultados son los obtenidos con la técnica Spin (S) y Spin-yield (SY). Con respecto a la estrategia a realizar ante la recepción de un mensaje, la técnica del coscheduling Dinámico (Dyn) es la que mejor se adapta a las diferentes cargas estudiadas. Por consiguiente, la mejor estrategia es la combinación de coscheduling Dinámico junto con la técnica de Block (Dyn-B), que se corresponde con la técnica del coscheduling Dinámico puro definido por Sobalvarro. Estos resultados coinciden con las conclusiones obtenidas por Gupta et al. en [GTU91] y Solsona en [Sol02]. En estos trabajos se demuestra que en entornos dedicados multiprogramados, el overhead añadido por el uso de primitivas de comunicación sin bloqueo no compensa el tiempo ahorrado por los cambios de contexto extras que se pueden producir por el uso de primitivas bloqueantes.

En conclusión, el uso de técnicas de coscheduling con control implícito minimiza el tiempo de espera de las aplicaciones distribuidas y, por tanto, maximiza el uso de los recursos disponibles; de manera que las aplicaciones paralelas pueden obtener, en algunos casos, un rendimiento comparable al proporcionado por un entorno de supercomputación dedicado. Asimismo, entre las diferentes propuestas de coscheduling con control implícito, las técnicas de coscheduling Dinámico son las que mejor se adaptan a las necesidades propias de los entornos cluster no dedicados; obteniendo un buen tiempo de ejecución para las aplicaciones paralelas con un mínimo overhead para el tiempo de respuesta de las locales. Sin embargo, cabe destacar que en todas las implementaciones descritas en entornos cluster no dedicados se ha restringido el grado de multiprogramación paralelo igual a uno ($MPL = 1$). Esta restricción en el grado de multiprogramación paralelo demuestra la necesidad de establecer mecanismos adicionales que garanticen el rendimiento de las tareas locales y permitan la ejecución de múltiples aplicaciones paralelas, simultáneamente. Con este objetivo, en esta tesis se presenta una nueva propuesta de coscheduling con control implícito, cuyos principios y objetivos son descritos en la próxima sección.

1.4. Una Nueva Propuesta de Coscheduling: Co-Scheduling Cooperativo

El progreso tecnológico desarrollado en los entornos cluster, junto con la reticencia al cambio por parte de los usuarios de esta misma tecnología, confluyen en el hecho de que cada vez la distancia entre las prestaciones ofrecidas por los equipos informáticos respecto de las necesidades reales de los usuarios sea mayor. Este progresivo aumento en los recursos de cómputo no utilizados

en una red de estaciones de trabajo invita a planteamientos más *agresivos,* respecto a los trabajos precedentes realizados en entornos cluster no dedicados, estudiando la viabilidad de ejecutar múltiples aplicaciones paralelas concurrentemente; es decir incrementando el grado de multiprogramación paralelo (MPL).

Este nuevo planteamiento comporta un amplio abanico de nuevos problemas a tener en cuenta. No se puede pasar por alto la reticencia de los usuarios locales de cada workstation a compartir su máquina simultáneamente con otros usuarios, dado que la ejecución de múltiples aplicaciones paralelas multiplica el riesgo de perturbar el trabajo local. Por tanto, la necesidad de implementar mecanismos que garanticen el rendimiento de las aplicaciones locales se convierte en un factor primordial. Ésto comporta que los recursos, tanto de CPU como de memoria, utilizados por las aplicaciones paralelas, deben adaptarse dinámicamente a las necesidades de los usuarios locales. Esta fluctuación en el número de recursos de cómputo disponibles dificulta, en gran medida, la coordinación entre procesos cooperantes a lo largo del cluster; lo que comporta que las técnicas de coscheduling tradicionales basadas en el análisis de los eventos de comunicación no sean suficientes para asegurar la progresión de múltiples aplicaciones paralelas en un cluster no dedicado.

Este nuevo escenario plantea una revisión del problema del coscheduling de las aplicaciones distribuidas en un cluster no dedicado. En nuestra opinión, el coscheduling no solamente debe gestionar *cuándo* deben ser asignados los recursos, objetivo alcanzado por las técnicas con control implícito tradicionales, si no que también *cuántos* recursos están disponibles para cada aplicación. Este doble propósito nos ha llevado a desarrollar una nueva propuesta de coscheduling, denominada *CoScheduling Cooperativo* (*CS*C), orientada a la coordinación de múltiples aplicaciones paralelas en un entorno no dedicado.

CSC, siguiendo el principio de coscheduling con control implícito, es un sistema totalmente distribuido (ver figura 1.9). De este modo, los recursos de cómputo de cada nodo son gestionados por el coscheduling cooperativo residente en el mismo, aprovechando los servicios que ofrece el s.o. subyacente. Estos recursos, de acuerdo con la doble funcionalidad atribuida a CSC, son asignados tanto en función de la ocurrencia de determinados eventos locales como de la recepción de aquellos eventos ocurridos en nodos remotos y que han modificado los recursos asociados a procesos cooperantes. La figura 1.9 ilustra, en flechas continuas, todos los eventos recibidos por el coscheduling cooperativo, así como todos aquellos subsistemas que gestiona (flechas discontinuas). En las siguientes secciones se describen los eventos analizados por CSC: la necesidad de cada uno de ellos y las decisiones de planificación tomadas en función de los mismos. Con objeto de facilitar esta descripción,

estos eventos han sido agrupados en dos grupos, los *eventos locales* y los *eventos remotos*. Finalmente, en la última sección se describirán todos aquellos objetivos planteados en el desarrollo de CSC.

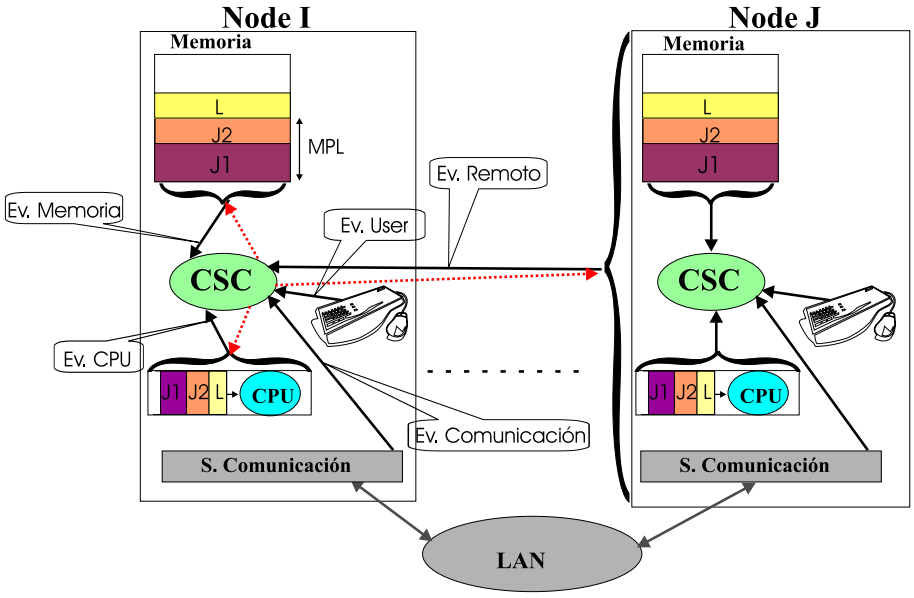

Figura 1.9: Eventos analizados por el CoScheduling Cooperativo.

1.4.1. CoScheduling Cooperativo: análisis de los eventos locales

Como se ha puesto de manifiesto a lo largo del presente capítulo, el uso de un cluster no dedicado para la ejecución de aplicaciones paralelas conlleva que el sistema operativo de cada máquina sea capaz de adaptarse dinámicamente a las necesidades de los diferentes usuarios que hacen uso de la misma. De acuerdo con el trabajo realizado por Arpaci et al. [ADV+95], el usuario local se caracteriza por unos elevados requerimientos de interactividad, mientras que en el usuario de las aplicaciones paralelas priman los requerimientos de cómputo y de comunicación. Conjugar las necesidades de ambos usuarios en un mismo sistema es el reto a perseguir por CSC.

Con este fin, CSC captura los siguientes eventos locales en cada uno de los nodos que integran el cluster:

- **Eventos de Actividad de Usuario Local (User)**. CSC debe adaptar su funcionamiento en función de los usuarios que estén presentes

en la máquina. En concreto, debe distinguir entre tres estados diferentes: máquina dedicada al usuario local, máquina dedicada al usuario de las aplicaciones paralelas o bien máquina compartida entre ambos usuarios. Con este fin, CSC captura información respecto la existencia de algún usuario local trabajando en dicha máquina.

- **Eventos de Comunicación**. Acorde con el propósito de las técnicas de coscheduling tradicionales, CSC debe garantizar, en la medida de lo posible, que las tareas cooperantes sean coplanificadas en función de sus necesidades de comunicación y de este modo minimizar el tiempo de espera de los procesos cooperantes que se comunican entre si. Esto comporta la captura, por parte de CSC, de todos aquellos eventos de comunicación asociados a cada una de las tareas paralelas presentes en cada nodo.

- **Eventos de CPU**. CSC tiene que garantizar un mínimo de recursos de CPU al usuario local, que se correspondan con sus necesidades reales, de manera que la cantidad de CPU asignada al usuario local sea suficiente para que su trabajo no se vea perturbado por la ejecución de las tareas paralelas. Asimismo, CSC debe asignar la CPU en el momento preciso para asegurar que las tareas cooperantes que se comunican entre si sean coplanificadas. Ambos hechos comportan que CSC debe conocer en todo momento el estado de la CPU, con objeto de poder asignar dinámicamente tanto la longitud del quantum como la prioridad de planificación, en función de las necesidades de ambos tipos de usuarios.

- **Eventos de Memoria**. En entornos monoprocesador, el sistema de paginación bajo demanda mejora la utilización tanto de la CPU como de la memoria al permitir que solamente un subconjunto de código y datos de cada proceso estén residentes en la memoria principal; permitiendo aumentar, por consiguiente, el número de procesos diferentes residentes en memoria [Den68, Den80]. Sin embargo, en entornos distribuidos, como es el caso de una NOW, los tradicionales beneficios de la paginación en entornos monoprocesador pueden verse deteriorados dependiendo de varios factores, como son: los patrones de sincronización de las aplicaciones paralelas, el tipo de localidad de memoria de las aplicaciones distribuidas, así como la interacción con las políticas de planificación de CPU. De este modo, el rendimiento de una política de coscheduling se puede reducir drásticamente si los requerimientos de memoria no son tenidos en cuenta [BHMW96, CNSW97, BF00, LKK00]. Con objeto de paliar este efecto, CSC monitoriza el estado de la memoria para poder reasignarla en función tanto de las necesidades de las tareas

locales como de las distribuidas. Asimismo, al igual que con la CPU, CSC debe garantizar al usuario local la memoria que éste necesite para el correcto progreso de su trabajo.

El análisis de los anteriores eventos permitirá a CSC asignar dinámicamente los recursos de CPU y memoria, en función tanto de las necesidades de los usuarios locales como de los de las aplicaciones distribuidas. Además, en función del grado de ocupación de dichos recursos, CSC podrá variar dinámicamente el grado de multiprogramación paralela (MPL), de modo que éste se ajuste en todo momento a la disponibilidad de dichos recursos.

Sin embargo, las decisiones locales tomadas por CSC deben ser tomadas de un modo coordinado a lo largo de todo el cluster teniendo en cuenta todos aquellos eventos ocurridos en los nodos remotos. En la siguiente sección se describe la necesidad de obtener dichos eventos remotos.

1.4.2. CoScheduling Cooperativo: análisis de los eventos remotos

Una aplicación está ejecutándose de una manera coordinada cuando todos los procesos cooperantes que la integran están ejecutándose simultáneamente en diferentes procesadores y disponen de la misma prioridad, en la asignación de recursos, a lo largo del cluster. Este comportamiento se ilustra en la figura 1.10.

La figura 1.10 muestra un cluster no dedicado formado por cinco nodos, en donde se ejecutan dos aplicaciones distribuidas, indicadas como $J1$ y $J2$ respectivamente, junto con dos aplicaciones locales. Con objeto de simplificar el ejemplo, asumiremos que todas las tareas que integran una misma aplicación distribuida requieren la misma cantidad de recursos de cómputo. En la figura 1.10(a), $J1$ solamente dispone del *20 %* de recursos de cómputo en los nodos 1 y 2 debido a que en dichos nodos hay presente un usuario local, el cual dispone de una mayor prioridad y por tanto tiene asignados una mayor cantidad de recursos de cómputo (*80 %*). En cambio, en los nodos 3 y 4, $J1$ y $J2$ tienen la misma prioridad y en consecuencia se reparten los recursos al *50 %,* de acuerdo con una política de asignación equitativa. Finalmente, el nodo 5 está dedicado exclusivamente a la ejecución de $J2$. En esta situación, $J1$ está privando a $J2$, en los nodos 3 y 4, de unos recursos de cómputo que, por otro lado, es probable que no necesite. Esto es debido a que los propios eventos de comunicación y sincronización con aquellas tareas cooperantes que disponen de menor cantidad de recursos provocarán que una aplicación (en este caso $J1$) progrese al ritmo marcado por las tareas ejecutadas en los nodos más lentos (nodo 1 y 2). Nuestra propuesta consiste en asignar, en la

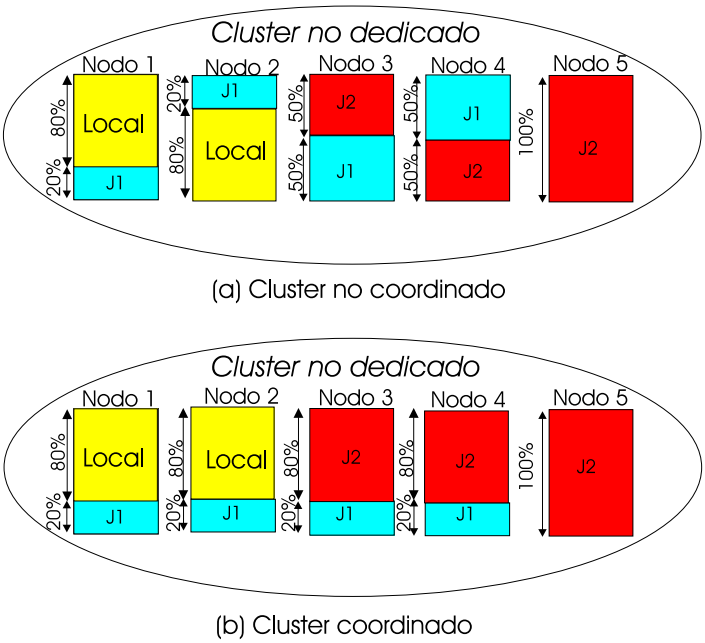

Figura 1.10: Ejemplo de coordinación en un cluster no dedicado.

medida de lo posible, la misma cantidad de recursos a todas las tareas que forman una aplicación distribuida, con el objetivo de que todas las tareas progresen uniformemente a lo largo del cluster. Esta situación se muestra en la figura 1.10(b). En este caso, $J1$ dispone, en cada uno de los nodos donde se ejecuta, de los mismos recursos asignados, mientras que $J2$, aunque no todas sus tareas disponen de los mismo recursos, si que la asignación de recursos a las mismas está mucho más equilibrada que con respecto el caso anterior. De esta manera, la asignación de los recursos se ajusta mucho más a las necesidades de las aplicaciones, de modo que todas las tareas que forman una misma aplicación distribuida progresarán al mismo ritmo y, por tanto, el rendimiento global del cluster se verá incrementado [FRS+97].

Este balanceo de recursos entre los procesos que forman una misma aplicación distribuida no puede ser alcanzado utilizando exclusivamente la información proporcionada por el nodo local, si no que es necesario el intercambio de información entre los nodos remotos cooperantes. En concreto, todos los eventos locales ocurridos en un nodo que provoquen una reasignación de los recursos de cómputo asociados a una tarea distribuida, deben ser notificados al resto de nodos que tienen tareas pertenecientes a la misma aplicación penalizada por dicha reasignación.

Por consiguiente, CSC toma sus decisiones de acuerdo con la información,

obtenida tanto localmente, como con la información externa proporcionada por los nodos remotos cooperantes.

1.4.3. Objetivos del CoScheduling Cooperativo

El análisis de los eventos descritos en las secciones anteriores permite a CSC asignar los recursos de cómputo, no solamente, cuando las correspondientes tareas los reclaman, si no también asignar tantos recursos como éstas necesiten. De acuerdo con este doble propósito, las decisiones de CSC están orientadas a alcanzar los siguientes objetivos:

- **Interacción entre los trabajos locales y paralelos**. CSC debe garantizar un mínimo de recursos de CPU y memoria al usuario local, de manera que la cantidad de recursos reservados sean suficientes para garantizar el rendimiento de sus trabajos dentro de unos límites aceptables por el mismo. Asimismo, CSC debe garantizar que estos recursos estén disponibles siempre que el usuario local los necesite, con objeto de mantener un buen tiempo de respuesta de sus aplicaciones interactivas. De acuerdo con la descripción realizada en la sección 1.3 del estado del arte, diferentes trabajos previos enmarcados en el área del coscheduling con control implícito han tratado este problema centrado exclusivamente en la asignación de la CPU, hecho que ha comportado una limitación en el grado de multiprogramación paralela igual a uno ($MPL = 1$) para garantizar el rendimiento de las aplicaciones locales, limitando de esta manera la eficiencia en el uso de los recursos disponibles. CSC afronta esta problemática para ambos subsistemas, CPU y memoria, con el objetivo de incrementar el grado de utilización de los recursos disponibles a lo largo del cluster. La consecución de este objetivo supone realizar un estudio previo exhaustivo respecto del grado de multiprogramación paralela soportado por un usuario local y como estos recursos disponibles son distribuidos en función de dicho grado y de los requerimientos de ambos tipos de aplicaciones, locales y paralelas. Como consecuencia de este estudio, se propondrá el mecanismo utilizado por CSC para asignar los recursos de cómputo.

- **Preempción de los procesos cooperantes de acuerdo con sus necesidades de comunicación**. Los trabajos previos en el área del coscheduling han mostrado que el rendimiento de las aplicaciones distribuidas depende, en gran medida, del grado de coscheduling alcanzado entre aquellas tareas con necesidades de comunicación/sincronización. Con este fin, CSC, al igual que las técnicas de coscheduling tradicionales, debe asegurar, en la medida de lo posible, la coplanificación de

las tareas que se comunican entre si, independientemente del número y tipo de aplicaciones, tanto locales como distribuidas, que compiten por los mismo recursos; garantizando, de este modo, un buen tiempo de ejecución para las aplicaciones distribuidas.

- **Asignación dinámica del quantum de tiempo**. El rendimiento de las aplicaciones paralelas en un entorno de tiempo compartido es muy sensible a la longitud de la rebanada de tiempo o quantum [FPCF01, YJ01]. En un entorno cluster no dedicado, la duración del quantum es un compromiso. Teniendo en cuenta las necesidades del usuario local, el quantum debería ser suficientemente corto para no degradar el tiempo de respuesta de las aplicaciones interactivas, mientras que desde el punto de vista del usuario paralelo, un quantum excesivamente corto podría degradar el rendimiento de la memoria cache, dado que cada vez que un proceso reanuda su ejecución deberá recargar de nuevo su working set. Esto significa que para amortizar el costo del cambio de contexto, el s.o. debería de garantizar que la longitud del quantum es suficientemente larga para recargar el working set y poderlo reutilizar [SRD01, MB91]. Sin embargo, un quantum excesivamente largo podría degradar el rendimiento de las técnicas de coscheduling dinámico [YJ01, FPCF01]. Por tanto, la longitud apropiada del quantum depende de las características de los trabajos concurrentes y ésta cambia de acuerdo con la variación de los requerimientos de comunicación y memoria de los mismos. Estas consideraciones determinan que una estrategia de coscheduling eficiente conlleve una asignación dinámica del quantum de tiempo. Esta nueva propuesta comporta la realización de un análisis detallado del rendimiento, tanto de las aplicaciones distribuidas como de las locales, en función de la longitud del quantum. Además, el rendimiento de la cache en función de diferentes longitudes de quantum, así como el grado de sensibilidad de las tareas distribuidas con respecto a dicho rendimiento deberá ser tenido en cuenta en nuestra nueva propuesta de coscheduling.

- **Coscheduling con requerimientos de memoria.** Tanto el comportamiento dinámico de los usuarios locales, como una política de mapping sin tener en cuenta las consideraciones de memoria, no pueden garantizar que los trabajos paralelos tengan suficiente memoria residente para encajar su working set, de manera que la memoria principal se desborde y, por consiguiente, se active el mecanismo de intercambio de páginas. Estos fallos de página pueden provocar una disminución en el rendimiento de las aplicaciones distribuidas, el cual se verá

incrementado para aplicaciones con elevados requerimientos de comunicación y sincronización [BHMW96]. Este comportamiento es debido a que cada fallo de página puede provocar retardos en cascada en los otros nodos donde se están ejecutando los procesos cooperantes. Este hecho comporta que una estrategia eficiente de coscheduling tenga en cuenta los requerimientos de memoria, tanto de las aplicaciones locales como de las distribuidas, con el objetivo de minimizar el número de fallos de página a lo largo del cluster. En trabajos previos desarrollados en el campo del coscheduling con control explícito se han propuesto diferentes algoritmos de control de admisión de trabajos en entornos cluster/multiprocesador dedicados para limitar los recursos de memoria consumidos por las aplicaciones paralelas [BF00, MZ95, PS96]. En nuestro conocimiento, CSC es la primera propuesta de coscheduling con control implícito que tiene en cuenta consideraciones de memoria. La obtención de este objetivo ha comportado la realización de un estudio sobre los requerimientos de memoria de un amplio abanico de aplicaciones distribuidas, así como del grado de repercusión de la paginación en el rendimiento de tareas distribuidas con diferente granularidad de comunicación. Atendiendo a los resultados de estos estudios, CSC integra el análisis de los eventos de memoria, tanto locales como remotos, con el objetivo de minimizar y controlar el impacto de una paginación excesiva.

- **Balanceo de los recursos de cómputo asignados a las diferentes tareas que forman una aplicación paralela**. El coscheduling, entendido de la manera tradicional, no es suficiente para asegurar la progresión de múltiples aplicaciones paralelas en un cluster no dedicado. Ésto es debido a que las tareas que forman una misma aplicación paralela deben disponer de una prioridad similar, en la asignación de recursos, en cada uno de los nodos donde se ejecutan, de lo contrario unas tareas progresarán más rápido que otras, lo que comportará una descoordinación global y, por tanto, un deficiente rendimiento de la aplicación distribuida. Con objeto de paliar este comportamiento, CSC asigna los recursos de cómputo a todas las tareas que integran una misma aplicación de un modo coordinado, gracias al conocimiento que tiene cada nodo respecto del estado de las tareas cooperantes que residen en nodos remotos. Asimismo, esta asignación de recursos, por parte de CSC, deberá de tener en cuenta el grado de heterogeneidad de los recursos de cómputo, tanto de CPU como de memoria, presentes a lo largo del cluster.

Los objetivos anteriormente descritos derivan implícitamente en los objetivos planteados en la realización de este trabajo. De este modo, a lo largo de esta tesis, se describirá como CSC alcanza cada uno de los objetivos anteriores, analizando la influencia de los mismos en el rendimiento tanto de las tareas distribuidas como de las locales y evaluando la viabilidad de su posible implementación. De este modo, la interacción entre todos estos objetivos dan lugar a esta nueva propuesta de coscheduling que permite planificar eficientemente múltiples aplicaciones distribuidas a lo largo de un cluster, aprovechando aquellos recursos no utilizados por los usuarios locales.

Capítulo 2

Viabilidad de la Computación Paralela en un Entorno Cluster no Dedicado Multiprogramado

Aspectos tales como los requerimientos de memoria de las aplicaciones, tanto locales como paralelas, la coplanificación de las tareas que se comunican entre si, la longitud del quantum, la presencia de usuarios locales en determinados nodos del cluster y el balanceo de recursos de cómputo a lo largo del cluster determinan totalmente el rendimiento de este tipo de entornos. En consecuencia, deben ser tenidos en cuenta por las correspondientes políticas de planificación tomadas en cada uno de los nodos del cluster. Con objeto de evaluar el peso de cada uno de estos factores, tanto en el rendimiento de las tareas distribuidas como de las locales, en este capítulo se analizarán, tanto en un entorno simulado como en un entorno real controlado, la influencia de dichos factores en un entorno cluster no dedicado multiprogramado.

Con la idea de enmarcar nuestro trabajo, en primer lugar se describirá el modelo de cluster no dedicado sobre el cual se desarrollarán todos nuestros estudios. A continuación, se explicará el entorno de simulación empleado, junto con las métricas utilizadas en nuestro estudio. En la siguiente sección se evaluará, por medio de la simulación, el rendimiento de las dos técnicas de coscheduling con control implícito tradicionales más utilizadas en la literatura, el coscheduling Dinámico y el coscheduling Implícito, con respecto a los factores enumerados anteriormente. Junto a estas técnicas se evaluarán diferentes modificaciones de las mismas, fruto tanto de diferentes referencias de la literatura como de nuestras propias intuiciones. Finalmente, la experimentación realizada por medio de la simulación será completada con un estudio hecho en un entorno real controlado sobre la interrelación entre la longitud del quantum y la memoria cache, así como su influencia en el

rendimiento de las tareas distribuidas y de las locales.

2.1. Modelo de un Sistema Cluster no Dedicado

En esta sección se describirá el modelo de cluster sobre el cual se desarrollarán nuestras propuestas. Este modelo es una extensión del presentado por Solsona et al. en [SGHL01a]. A diferencia del modelo original, donde cada nodo era modelado exclusivamente mediante el subsistema de comunicación y de planificación de la CPU, el modelo utilizado en esta tesis también contempla el correspondiente subsistema de memoria, la actividad asociada al usuario local (teclado y ratón), así como la información de sistema procedente de nodos remotos.

Con objeto de simplificar el modelo, tanto como sea posible, varias suposiciones son realizadas. Asimismo, con la idea de facilitar la posterior implementación de nuestras propuestas en el s.o. Linux [BC01], algunas de estas suposiciones son basadas en propiedades que presentan cualquier s.o. *UNIX* [Bac86], en general, y por tanto, *Linux* en particular. Estas suposiciones son las siguientes:

- Una aplicación distribuida i (Job_i) de tamaño m está compuesta por un conjunto de m tareas ($task$), las cuales ya se encuentran mapeadas en los n nodos que constituyen el cluster.
- Cada una de las tareas que constituyen un mismo trabajo será mapeada en un nodo distinto del cluster.
- Cada nodo k del cluster ($node_k$) tendrá mapeadas, en cada instante de tiempo, *MPL* ("*MultiProgramming Level*") tareas distribuidas.
- Cada $node_k$ se caracterizará por ser un entorno monoprocesador con un sistema operativo de tiempo compartido y con un algoritmo de planificación, por defecto, *Round-Robin* apropiativo.
- En cada $node_k$ se asumirá un subsistema de memoria virtual basado en la paginación bajo demanda.
- Nuestro modelo asume que cada nodo del cluster está bajo el control de nuestro esquema de coscheduling y que todos los nodos del cluster disponen del mismo s.o. Es importante destacar que esta restricción, de acuerdo con los diferentes estudios de carga realizados en entornos

NOW [ACP+95, AES97, Ryu01], se corresponde con la realidad que presentan la mayoría de laboratorios de PCs en cualquier institución o empresa.

En esta sección se describirá como se ha modelizado cada nodo del cluster. Con objeto de facilitar su comprensión, la descripción de nuestro modelo se ha dividido en dos partes diferentes: el *modelo del coscheduling tradicional*, formado por el subsistema de planificación de CPU y de comunicación, y el *modelo ampliado*, el cual contempla el subsistema de memoria, la interactividad del usuario local y el intercambio de información de sistema entre nodos cooperantes. La figura 2.1 presenta un esquema simplificado de la organización de nuestro modelo.

2.1.1. Modelo de coscheduling tradicional

Como se puede ver en la figura 2.1, el planificador de cada nodo del cluster dispone de una cola de tareas preparadas para ejecución, denominada *cola de preparados* (RQ), donde las tareas son ordenadas de acuerdo con su prioridad. El *top* de la cola ($RQ[0]$) corresponde a la tarea actualmente en ejecución mientras que el *bottom* ($RQ[\infty]$), corresponde a la tarea preparada para ser ejecutada con menor prioridad. Asimismo, dispone de una *cola de espera* (WQ), donde se almacenan aquellas tareas que se encuentran bloqueadas en espera de la ocurrencia de un determinado evento (p.e: un evento de comunicación o bien un fallo de página de memoria).

El planificador de tiempo compartido divide el tiempo de CPU en *épocas*, de manera que al comienzo de cada época, a cada tarea se le asigna un quantum de tiempo, durante el cual la tarea podrá ser ejecutada en la CPU. Cuando la tarea en ejecución ha expirado su tiempo o bien se ha quedado bloqueada en espera de un determinado evento de E/S, el siguiente proceso en la cola de preparados será elegido para ser ejecutado. La época finaliza cuando todos los procesos en la RQ han finalizado su correspondiente quantum.

Con respecto al subsistema de comunicación, cada nodo dispone de dos colas denominadas de *Recepción de Mensajes (RMQ) y Envío de mensajes (SMQ)*, donde se almacenan tanto los mensajes enviados a nodos remotos como los mensajes recibidos desde los mismos. Es importante destacar que la mayoría de sistemas UNIX con el protocolo de comunicación TCP(UDP)/IP disponen de colas semejantes.

El s.o. de cada $node_k$ mantiene para cada tarea i ($task_i$), ejecutándose en dicho nodo, la siguiente información:

- $task_i.q_n$. Quantum de tiempo asignado a $task_i$ en la enésima época. Nuestro modelo contempla que cada tarea tiene asociado un quantum

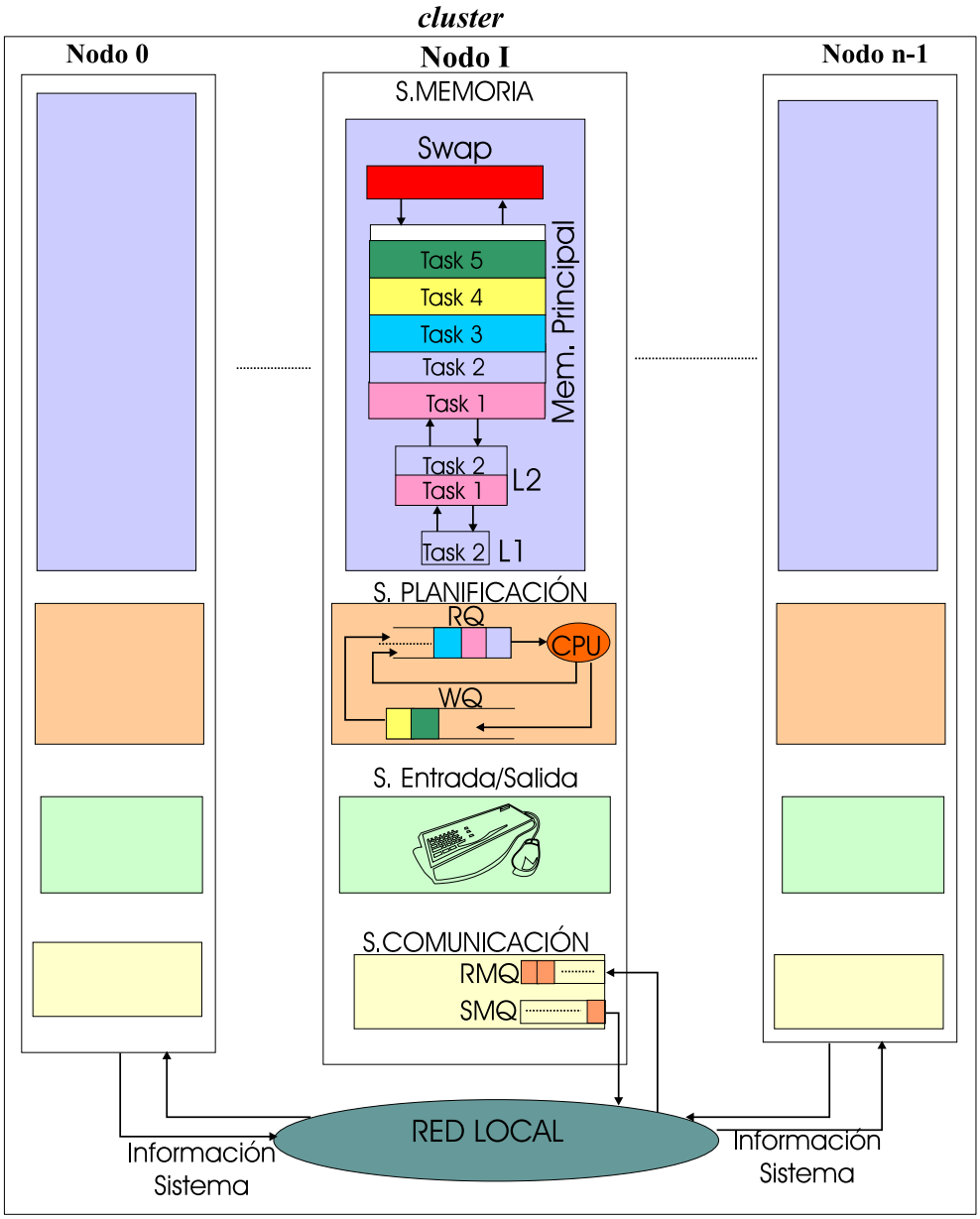

Figura 2.1: Modelo de un cluster no dedicado.

dinámico ($task.q_n$), el cual es decrementado durante cada época. Al inicio de cada época, un nuevo quantum es reasignado. Por defecto, nuestro modelo asume que todas las tareas tienen la misma longitud de quantum, la cual será indicada como $DEF\_QUANTUM$.

- $task_i.tc$. Número de ciclos de ejecución consumidos por $task_i$ desde el inicio de su ejecución.

- $task_i.tr$. Número de ciclos transcurridos desde el lanzamiento de $task_i$. Al final de la ejecución, este campo se corresponde con el tiempo de retorno de $task_i$.

- $task_i.trq$. Número de ciclos medios consumidos por $task_i$ en la cola de preparados en espera a ser planificada durante cada época.

- $task_i.rec$. Número de mensajes recibidos en la cola de recepción (RMQ) para $task_i$.

- $task_i.send$. Número de mensajes enviados por $task_i$ a la cola de envío (SMQ).

- $task_i.com$. Número de mensajes enviados y recibidos por $task_i$. Este valor se calculará a partir de la suma de los dos campos anteriores:

$$task_i.com = task_i.send + task_i.rec \tag{2.1}$$

- $task_i.de$. Número de veces que $task_i$ ha sido adelantada en la cola de preparados por otra tarea, desde la última vez que dicha tarea fue insertada en la cola RQ. Este campo será utilizado para evitar la inanición de las tareas locales y de las tareas distribuidas con escasa comunicación.

Es importante destacar que a excepción del campo *de*, la mayoría de campos enumerados, son mantenidos por gran parte de los sistemas UNIX actuales. Por citar un ejemplo, en Linux, cada tarea se representa mediante un bloque de control de procesos ("*Process Control Block (PCB)*"), el cual contempla explícita o bien implícitamente la mayoría de los anteriores campos.

2.1.2. Modelo ampliado

Como se puede ver en la figura 2.1, el subsistema de memoria de cada $node_k$ está formado por dos niveles de cache (L1 y L2) junto con una memoria principal y una memoria swap.

Con respecto a la memoria cache, se asume que su contenido se va actualizando dinámicamente con los bloques de datos pedidos por el proceso en ejecución. Teniendo en cuenta las políticas utilizadas por la mayoría de procesadores comerciales, se asume una política de reemplazo LRU ("*Least Recently Used*").

La gestión de la memoria principal se realiza mediante un algoritmo de paginación bajo demanda. De esta manera, si la página referenciada por una tarea no se encuentra en su conjunto residente ("*resident set*") se producirá un fallo de página. Este fallo comportará que dicha tarea sea suspendida en la cola de espera (WQ) hasta que la página demandada sea cargada en su conjunto residente.

El algoritmo de reemplazamiento de páginas será aplicado a todas las páginas de memoria, independientemente de la tarea propietaria de las mismas. De este modo, el tamaño del conjunto residente de cada tarea podrá variar aleatoriamente a lo largo del tiempo. Teniendo en cuenta las tendencias de los actuales s.o. UNIX, se asume una política de reemplazo de página LRU ("*Least Recently Used*"). En dicho algoritmo se reemplaza aquella página que no ha sido utilizado durante el periodo de tiempo más largo. Un ejemplo del mismo, es el s.o. Linux, que utiliza el algoritmo de reemplazo por segunda oportunidad, el cual es una aproximación a la política LRU.

Con respecto a la memoria swap y con objeto de simplificar el diseño de nuestros algoritmos se asume un tamaño infinito del mismo. En el capítulo de experimentación se estudiará el peso de esta restricción con respecto al rendimiento de nuestras propuestas.

Asimismo, este nuevo modelo contempla la información de la actividad del usuario local, mediante el análisis de la ocurrencia de los eventos de teclado y ratón. Finalmente, con objeto de coordinar las decisiones de asignación de recursos tomadas en cada uno de los nodos que integran el cluster se intercambia información de sistema, referente al estado de la memoria y a la actividad del usuario local, entre aquellos nodos que cooperan entre si para la ejecución de una determinada aplicación distribuida.

El s.o. mantiene para cada $task_i$ la siguiente información referente a su estado y a sus requerimientos de memoria:

- $task_i.vmem$. Tamaño de memoria requerido por $task_i$.
- $task_i.rss$. Tamaño de memoria residente de $task_i$.
- $task_i.flt(nrpg\_fault)$. Número medio de fallos de página realizados por $task_i$ durante un segundo (durante toda la ejecución). Cabe destacar que este intervalo de tiempo ($= 1s$) coincide con el periodo tomado por el *swapper* de algunos s.o. UNIX para comprobar la necesidad de intercambiar páginas con el área de swap.
- $task_i.uid$. Identificador de usuario de $task_i$. Con objeto de simplificar la identificación de los trabajos locales y paralelos, todas las aplicaciones paralelas se ejecutarán bajo un mismo usuario genérico, el cual será

identificado como *PARAL,* mientras que las locales pertenecerán al usuario genérico *LOCAL*.

- $task_i.jid$. Cada tarea paralela tiene asociado un identificador correspondiente a la aplicación paralela a la que pertenece. Este identificador se irá incrementando sucesivamente a medida que se vayan lanzando nuevos trabajos al sistema. Por tanto, el trabajo con el identificador mayor es el último que ha sido lanzado al sistema. Cabe destacar que las dos librerías de paso de mensajes de uso más extendido, PVM y MPI, trabajan con este tipo de identificadores, de manera que agrupan todas las tareas que pertenecen a una misma aplicación distribuida bajo un mismo identificador común.

- $task_i.state$. Este campo, solamente utilizado por las tareas distribuidas, se corresponde al estado asociado con el trabajo paralelo al que pertenecen, de modo que todas las tareas que pertenecen al mismo trabajo tendrán el mismo estado. En concreto, este campo puede tener los siguientes tres valores:

  - LOCAL. Indica que una o varias de las tareas tiene los recursos de cómputo limitados debido a la presencia de un usuario local. Asociado a dicho estado, cada tarea distribuida dispone de un campo denominado $task_i.nr\_locals$, el cual indica el número de tareas de la aplicación (a la cual pertenece $task_i$), que tienen reducidos sus recursos de cómputo debido a la presencia de un usuario local en su máquina.
  - STOP. Indica que una o varias de las tareas están paradas debido a que la memoria se ha desbordado en los nodos en los que residen.
  - NULL. En caso de que no se cumplan ninguna de las dos condiciones enumeradas anteriormente, este campo toma este valor. Asimismo, todas las tareas locales tomarán este valor.

Asimismo, el s.o. de cada $node_k$ mantiene la siguiente información global respecto al subsistema de memoria, actividad de usuario local y estado del resto de nodos cooperantes:

- $node_k.mr_n$. Ratio de fallos de la cache L2, calculada de acuerdo con la siguiente expresión:

$$mr_n = \frac{L2\_cache\_misses_n}{L1\_cache\_misses_n}, \tag{2.2}$$

donde $L2/L1\_cache\_misses_n$ son el número de fallos de la cache L2 y L1 respectivamente, producidos durante la enésima época. Aunque este parámetro no es proporcionado por la mayoría de s.o. comerciales, se puede obtener ya sea a partir de los contadores hardware que disponen la mayoría de microprocesadores actuales [Pet02] o bien mediante el uso de métodos analíticos [SRD01].

- $node_k.M$. Tamaño de la memoria principal del $node_k$.
- $node_k.mem$. Suma de los requerimientos de memoria de todas las tareas que se están ejecutando en el $node_k$.
- $node_k.mem\_par$. Suma de los requerimientos de memoria de todas las tareas paralelas que se están ejecutando en el $node_k$.
- $node_k.LOCAL\_USER$. Variable booleana, que toma el valor uno cuando en el $node_k$ se incorpora a trabajar el usuario local. Se asume que el sistema toma como usuario local todo aquel usuario de la máquina que haga un uso interactivo de la misma, es decir, que haga uso del teclado o bien del ratón.
- $node_k.MPL$. Número de tareas distribuidas que se ejecutan concurrentemente en el $node_k$.
- $node_k.MPL\_TOTAL$. Número de tareas, tanto locales como distribuidas, que se ejecutan concurrentemente en el $node_k$.
- $node_k.L$. Porcentaje de recursos de cómputo, CPU y memoria, asignado a aquellas tareas distribuidas residentes en nodos con actividad de usuario local. Este porcentaje es uniforme a lo largo del cluster.
- $node_k.cooperating(task_i)$. Cada nodo gestiona una lista, denominada *cooperating*, que contiene las direcciones de todos aquellos nodos del cluster donde se encuentran aquellas tareas cooperantes (el resto de tareas distribuidas pertenecientes a la misma aplicación distribuida) de la tarea $task_i$ que se está ejecutando en dicho nodo. Cabe remarcar que, aunque la mayoría de sistemas operativos actuales no disponen de este tipo de información, ésta se puede obtener ya sea mediante una monitorización de las comunicaciones a nivel del núcleo o bien a partir de los mismos *daemons* de comunicación utilizados por las librerías de paso de mensajes; como por ejemplo PVM.

Es importante destacar que, salvo el parámetro $node_K.mem\_par$ y el parámetro $task_i.state$, el resto de campos mencionados son proporcionados por la

gran mayoría de los actuales s.o. UNIX, ya sea de una manera explícita o bien de una manera implícita.

2.2. Entorno de Simulación

Con objeto de verificar la viabilidad de nuestra propuestas, se ha utilizado una extensión del simulador presentado en [SGHL01b]. El simulador original permitía la simulación de diferentes políticas de coscheduling con control implícito, basadas en el modelo de coscheduling tradicional explicado en la sección anterior. En esta tesis, la capacidad de dicho simulador ha sido ampliada con las nuevas funcionalidades incorporadas en el modelo de coscheduling ampliado.

Nuestro simulador, basado en colas [Kle76], reproduce el modelo de cluster explicado en la sección anterior. Se ha simulado un cluster de n nodos homogéneos, donde cada nodo se ha simulado mediante un subsistema de planificación formado por una cola de preparados, una cola de espera y una CPU; un subsistema de memoria y un subsistema de comunicación. De este modo, cuando una tarea expira su quantum asignado, la tarea es sacada de la CPU y es reinsertada en la cola de preparados (RQ), siempre y cuando no haya finalizado su ejecución. Si una tarea no expira su quantum debido a la generación de un fallo de página o bien a la espera de un evento de recepción, ésta será insertada en la cola de espera (WQ). El planificador local fijará el orden de las tareas en la RQ de acuerdo con una política fijada por el usuario. Con respecto al modelo de memoria, cabe destacar que el simulador utilizado no tiene en cuenta los efectos de la memoria cache. En este sentido cabe remarcar que se ha preferido realizar el estudio sobre la influencia de la cache en un entorno cluster real, debido a las facilidades que incorporan los procesadores actuales para monitorizar el rendimiento de la memoria cache y a la existencia de drivers que permiten la realización de dicha monitorización desde el espacio de usuario.

El apéndice A muestra una descripción detallada de las diferentes funciones de distribución empleadas para la simulación del cluster no dedicado. La figura 2.2 muestra las principales entradas y salidas de nuestro simulador. El significado de las correspondientes entradas es el siguiente:

- **n (número de nodos del cluster)**. El valor, por defecto, de este parámetro es 8.
- **n_jobs (número de trabajos servidos por el cluster)**. Este parámetro fija el final de cada simulación. Todas las pruebas han sido realizadas con un valor de $n\_jobs = 10000$.

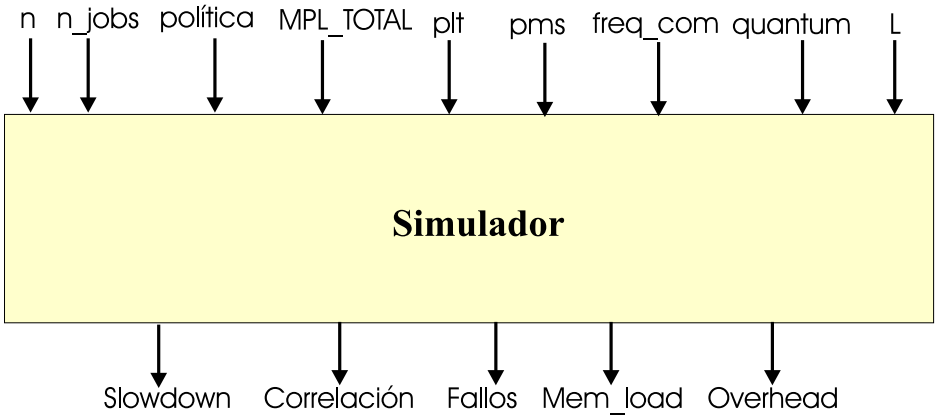

Figura 2.2: Entradas/Salidas del simulador.

- **política**. Esta entrada fija la política de planificación/coscheduling utilizada por el simulador. Nuestro simulador acepta tres políticas diferentes: Round-Robin (RR), una política de coscheduling Dinámico (DYN) o bien una política de coscheduling Implícito (IMPLI). En concreto, la política de coscheduling Dinámico implementada se corresponde con la versión implementada por Solsona et al.[Sol02], descrita en la sección 1.3.2. En esta implementación, cada vez que un proceso recibe un mensaje, su prioridad es incrementada por el propio planificador de manera que dicho proceso será planificado inmediatamente. Con objeto de no degradar excesivamente el rendimiento de las tareas locales se limita el número máximo de adelantos que pueden sufrir las tareas locales en espera a ser planificadas. El máximo número de adelantos ha sido fijado en dos, de acuerdo con el estudio mostrado en el apéndice B. La implementación realizada del coscheduling Implícito está basada en la definición dada por Arpaci et al. en [AD01] (ver sección 1.3.2). En la simulación realizada, el tiempo de espera (*spin*) ha sido fijado igual al tiempo que tarda un mensaje en realizar un *round-trip*. De acuerdo con el valor de la latencia de red tomada por el simulador, se ha definido un valor de tiempo de espera igual a $10u.t.$

- **MPL_TOTAL**. Este parámetro fija el grado de multiprogramación máximo por nodo. Cabe destacar que, este parámetro no distingue entre las tareas locales y distribuidas.

- **plt ("probability of local task")**. Cada vez que se genera un nuevo trabajo, este valor fijará la probabilidad de que sea distribuido (*1-plt*) o local (*plt*).

- **pms ("probability of memory size")**. Este valor fijará la probabilidad de que las tareas generadas tengan un alto (pms=1) o bien bajo requerimiento de memoria (pms=0). Se asume que todas las tareas que pertenecen a un mismo trabajo distribuido tienen asignado el mismo requerimiento de memoria. De este modo, se entiende que una tarea tiene un bajo requerimiento de memoria cuando la cantidad de memoria solicitada no supera el *25 %* del tamaño de la memoria principal, mientras que es alto en caso contrario. En el apéndice A se describe con detalle como se ha simulado el working set de cada tarea, así como el mecanismo de memoria virtual.

- **freq_com**. Este parámetro fija la frecuencia de emisión de mensajes para cada tarea distribuida. El patrón de comunicación empleado es de uno a todos. Se han simulado dos tipos de primitivas de envío: una primitiva de *envío no bloqueante*, de manera que la tarea emisora reanuda su ejecución tras el envío, y una primitiva de *envío bloqueante*, de manera que la tarea emisora queda bloqueada hasta que recibe el reconocimiento de todas las tareas cooperantes a las cuales ha enviado el mensaje.

- **quantum**. Este valor es la longitud media de quantum asociado a cada tarea. Por defecto se toma un valor igual a $100u.t.$

- **L (porcentaje de recursos asignados a las tareas paralelas)**. Este parámetro permite fijar el máximo porcentaje de recursos asignados a las tarea paralelas. Por defecto, este valor es tomado igual al *100 %*. Este valor implica que todas las tareas locales y distribuidas competirán por la asignación de recursos en igualdad de condiciones.

Respecto de las salidas, nuestro simulador proporciona dos métricas específicas sobre el estado de la memoria del cluster (*mem_load y Fallos* de Página) junto con tres métricas de rendimiento de las tareas, una específica para las tareas distribuidas (*Correlación*) y las otras dos comunes para ambos tipos de tareas (*Slowdown* y *Overhead* de respuesta). El significado de estas cinco métricas es el siguiente:

- **Overhead de respuesta (LOCALES/DISTRIBUIDAS)**. Esta métrica da el tiempo medio que consume una tarea (local o distribuida) en ser planificada desde la última vez que se insertó en la cola de preparados. Esta métrica se calcula a partir de la siguiente expresión:

$$Overhead = \frac{\sum_{i=1}^{n} \frac{node_i \cdot \sum_{k=1}^{node_i.TOTAL\_TASKS} task_k.trq}{node_i.TOTAL\_TASKS}}{n}, \qquad (2.3)$$

donde $node_i.TOTAL\_TASKS$ es el número de tareas ejecutadas en el $node_i$, $task_k.trq$ es el tiempo medio gastado por una tarea en la cola de preparados a la espera de ser planificada y n es el número de nodos del cluster. Esta métrica nos dará, para el caso de las tareas distribuidas, una medida del grado de coscheduling alcanzado, mientras que para las locales nos dará una medida del tiempo de respuesta de las mismas.

- **Slowdown (LOCALES/DISTRIBUIDAS)**. Este parámetro se define de acuerdo con la siguiente expresión:

$$Slowdown = \frac{\sum_{k=1}^{TOTAL\_JOBS} \frac{job_k.tr}{job_k.tc}}{TOTAL\_JOBS}, \tag{2.4}$$

donde $TOTAL\_JOBS$ es el número de trabajos de un determinado tipo (locales o distribuidos) generados a lo largo de la simulación, $job_k.tc$ es el tiempo de procesamiento de job_k y $job_k.tr$ es el tiempo de retorno, que coincidirá con el tiempo de retorno de la tarea más lenta del job_k. Con objeto de facilitar el análisis respecto el grado de coordinación alcanzado por las diferentes políticas evaluadas, el simulador asigna a todas las tareas de un mismo trabajo paralelo un mismo tiempo de procesamiento, hecho que comporta que $job_k.tc$ se corresponda con el tiempo de ejecución del job_k en un entorno cluster dedicado. Finalmente cabe observar que esta métrica servirá para comparar el comportamiento de las diferentes técnicas, tanto en lo que respecta al rendimiento de las tareas locales como al de las distribuidas, así como para estudiar los límites asociados con el grado de multiprogramación de las tareas distribuidas.

- **Correlación**. Este parámetro muestra el grado de coordinación entre las decisiones tomadas en cada uno de los nodos que constituyen el cluster. Esta métrica se define de acuerdo con la siguiente expresión:

$$Correlacion = 100 - \frac{\sum_{k=1}^{TOTAL\_JOBS} (\frac{Job_k.tr_{slow} - Job_k.tr_{fast}}{Job_k.tr_{slow}})}{TOTAL\_JOBS} \times 100, \tag{2.5}$$

donde $TOTAL\_JOBS$ es el número de trabajos paralelos ejecutados en el cluster durante la simulación, $Job_k.tr_{fast}$ y $Job_k.tr_{slow}$ son el tiempo de retorno asociado a la tarea más rápida y más lenta pertenecientes al Job_k, respectivamente. Es importante observar que de acuerdo con el modelo presentado, cada nodo tiene mapeada como máximo una tarea perteneciente a un job concreto.

- **Fallos** de Página. Este parámetro se calculará de acuerdo con la siguiente expresión:

$$Fallos = \frac{\sum_{i=1}^{n} \frac{node_i . \sum_{k=1}^{node_i.TOTAL\_TASKS} task_k.nrpg\_fault}{node_i.TOTAL\_TASKS}}{n}, \quad (2.6)$$

  donde n es el número de nodos del cluster, $task_k.nrpg\_fault$ es el número de fallos de página de $task_k$ y $node_i.TOTAL\_TASKS$ es el número de tareas, tanto locales como distribuidas, ejecutadas en el $node_i$. Este parámetro será calculado únicamente para validar el buen funcionamiento de las técnicas de coscheduling cuando los requerimientos de memoria sobrepasen el tamaño de la memoria principal.

- **mem_load**. Esta métrica muestra los requerimientos de memoria del conjunto de tareas ejecutadas en cada nodo. Es decir, un valor $mem\_load < 1$ significa que los requerimientos de memoria no superan el tamaño de la memoria, mientras que $mem\_load > 1$ implica que la memoria está desbordada. Este valor es una media calculada a partir de la carga de cada nodo.

Finalmente, cabe destacar que el objetivo de los diferentes algoritmos de coscheduling será minimizar las métricas de Overhead, *Fallos y Slowdown* y maximizar la métrica *Correlación.*

2.3. Análisis del Rendimiento de un Entorno Cluster no Dedicado Multiprogramado

En esta sección se analizarán las limitaciones del *coscheduling Dinámico* (DYN) e *Implícito* (IMPLI) en un entorno cluster no dedicado multiprogramado. Con objeto de obtener una cota mínima de coscheduling como referencia, se ha simulado una tercera alternativa caracterizada porque todos los nodos planifican sus tareas, independientemente del resto, de acuerdo con un algoritmo de planificación de *Round-Robin* (RR). Asimismo, se evaluarán diferentes modificaciones de dichas técnicas, asociadas a ideas aplicadas en entornos dedicados, descritas en trabajos anteriores a esta tesis, así como a nuestras propias intuiciones. Estas técnicas serán evaluadas bajo diferentes escenarios, que representan la alta variabilidad que caracteriza un cluster no dedicado. En concreto, esta sección se ha dividido en tres partes diferentes, atendiendo a diferentes escenarios asociados con el estado del subsistema de memoria. Las dos primeras subsecciones corresponden al estudio realizado

por medio de la simulación en función del grado de ocupación de la memoria principal. En la última subsección se ha analizado, en un entorno real, la influencia de la memoria cache en el rendimiento de las aplicaciones distribuidas en un entorno cluster dedicado/no dedicado multiprogramado.

2.3.1. Rendimiento de las aplicaciones locales y distribuidas en un entorno con bajos requerimientos de memoria

Para las tres políticas mencionadas (DYN, IMPLI y RR) se ha obtenido el slowdown, correlación y overhead de las tareas distribuidas en dos tipos de entornos diferentes: en el primero se ha fijado el número de tareas locales generadas (*plt=0,3*) y se ha variado el grado de multiprogramación desde uno a cinco (fig. 2.3(izq.)), y en el segundo se ha fijado el grado de multiprogramación en tres ($MPL\_TOTAL = 3$) y se ha variado el número de tareas locales generadas (fig. 2.3(der.)). Asimismo, en ambos entornos se ha impuesto la restricción de que todas las tareas, tanto locales como distribuidas, encajen en memoria. Estas pruebas permitirán evaluar la eficiencia en la gestión de la CPU por parte de las tres técnicas evaluadas, así como estudiar la influencia del grado de multiprogramación y el número de tareas locales en la probabilidad de coscheduling de cada una de las tres técnicas evaluadas.

El slowdown obtenido en ambos entornos (ver figura 2.3(arriba)) refleja como la técnica del coscheduling Dinámico obtiene el mejor resultado, corroborando los resultados obtenidos por otros autores [Ang00], mientras que la técnica de Round-Robin obtiene unos resultado muy pobres debido a la ausencia total de coordinación entre los nodos. El bajo rendimiento obtenido con la técnica de coscheduling Implícito, cercano al del Round-Robin en algunos casos, es debido a que al aumentar el grado de multiprogramación o bien el nombre de tareas locales competidoras, crece la probabilidad de que el proceso destino de una primitiva de comunicación no esté planificado en el nodo remoto y por tanto no pueda responder dentro del intervalo de espera activa, lo que comporta que la espera activa realizada por el proceso emisor resulte innecesaria. Un aspecto que no puede ser pasado por alto, es que para un valor de $MPL\_TOTAL = 5$, el slowdown obtenido con las tres técnicas evaluadas supera el grado de multiprogramación. Este resultado indica que los algoritmos de coscheduling evaluados no son capaces de coplanificar eficientemente las tareas distribuidas que comunican entre si por encima del mencionado umbral ($MPL\_TOTAL = 5$). Este último aspecto se refleja perfectamente en el incremento de la pendiente de la métrica de

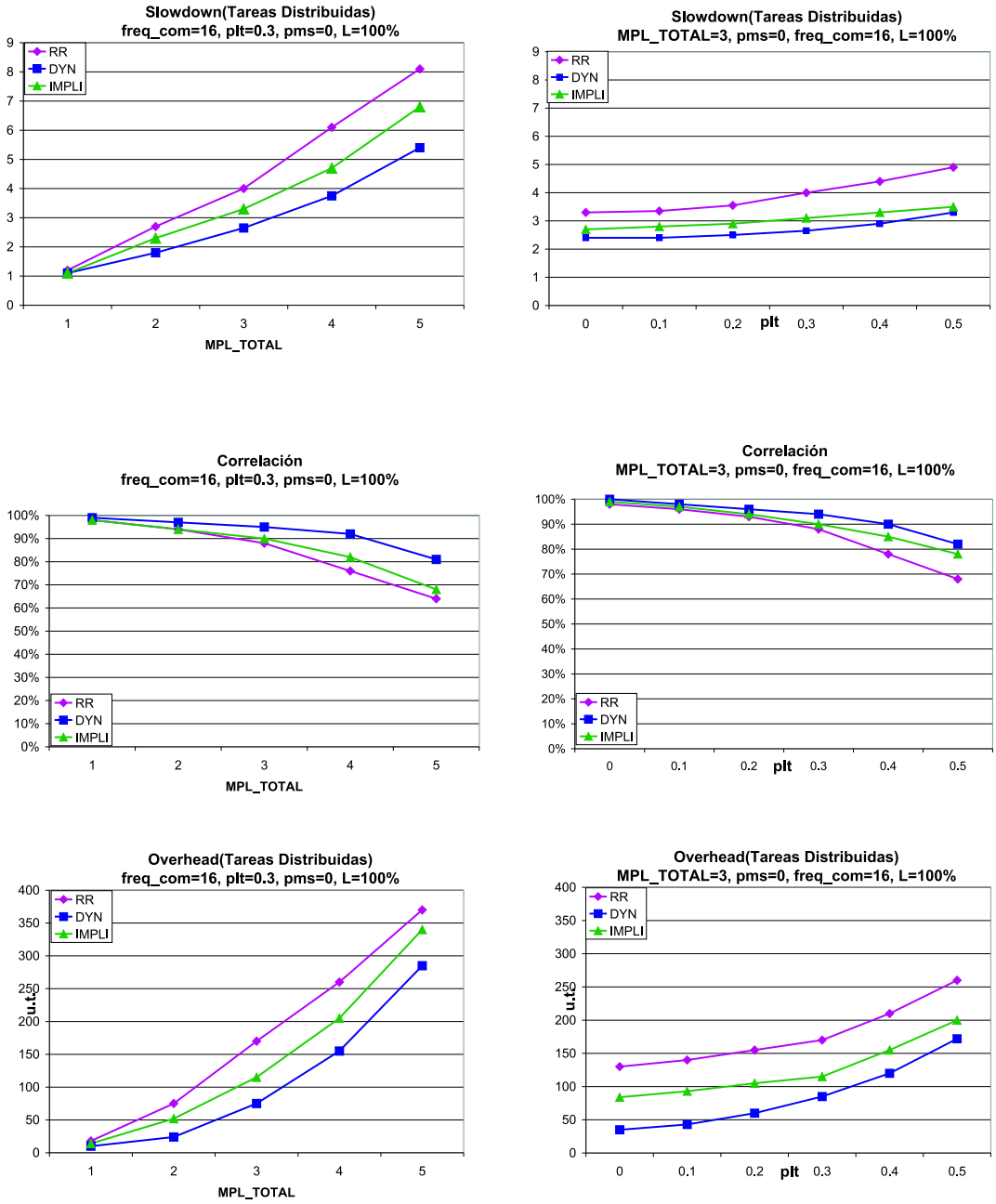

Figura 2.3: Rendimiento de las tareas distribuidas al variar el grado de multiprogramación (izquierda) y el número de tareas locales (derecha).

overhead y correlación para valores de $MPL\_TOTAL > 4$, tanto para el coscheduling Dinámico como para el Implícito. Un último aspecto a resaltar

es la disminución del rendimiento de la técnica del coscheduling Dinámico a medida que se aumentan el número de tareas locales en el sistema. Esta caída de rendimiento es debida a la limitación, que impone el coscheduling Dinámico implementado, en el número máximo de adelantos que puede sufrir una tarea local, denominada *condición de apropiación* e indicada con el acrónimo *MNO*, siendo en estas pruebas $MNO = 2$ (ver apéndice B). Este comportamiento es debido a que en este escenario se ha mantenido constante el valor de MPL_TOTAL y se ha variado el valor de la probabilidad de usuario local (plt) y, en consecuencia, el número de tareas locales generadas ha aumentado a medida que se ha incrementado el valor de *plt*. Este incremento en el número de tareas locales generadas ha comportado que se limite la posibilidad de que las tareas distribuidas, que se encuentran en la cola de RQ, puedan adelantar a las locales, y en consecuencia, puedan alcanzar el coscheduling con el resto de tareas cooperantes residentes en nodos remotos.

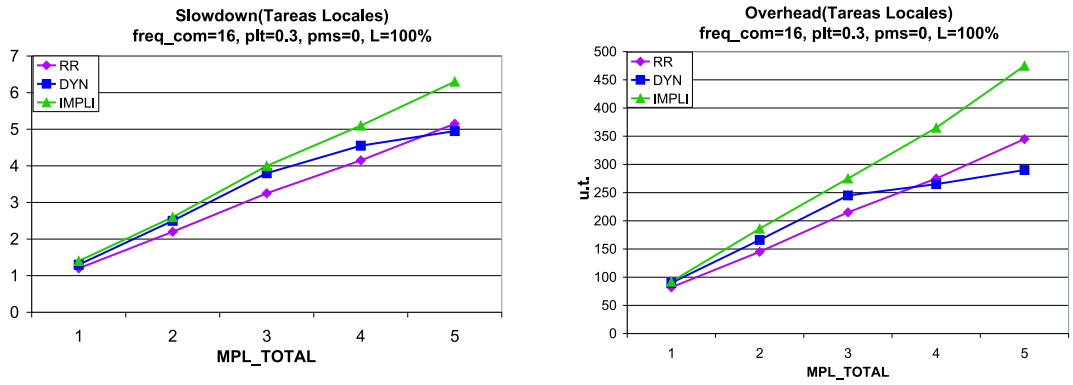

Figura 2.4: Slowdown y overhead de las tareas locales al variar el grado de multiprogramación.

Un objetivo primordial en nuestro estudio es no ralentizar en exceso el rendimiento de las tareas locales. Con objeto de comprobar esta premisa, se ha calculado el slowdown y el overhead de las tareas locales ejecutadas en el mismo escenario anteriormente descrito (figura 2.4). Cabe destacar que el estudio del slowdown nos dará una medida de cuánto se perjudicará una tarea local de cómputo intensivo, mientras que el overhead reflejará el tiempo de respuesta que cabría esperar para un usuario interactivo. Tal y como cabía esperar, las dos técnicas de coscheduling (DYN e IMPLI) infringen un slowdown y overhead superior que la técnica de RR, aunque el overhead producido por el coscheduling Dinámico es siempre inferior al producido por el Implícito. Ésto es debido a la limitación, anteriormente explicada, en el

número de adelantos impuesta por el coscheduling Dinámico. Este hecho se refleja en el análisis del overhead introducido por el coscheduling Dinámico; donde se observa como para un grado de multiprogramación superior a tres la pendiente del overhead se suaviza, garantizando un mínimo tiempo de respuesta para un hipotético usuario interactivo. Sin embargo, tanto el slowdown como el overhead obtenido para un $MPL\_TOTAL > 2$ pueden ser considerados demasiado grandes atendiendo a las necesidades de un usuario local.

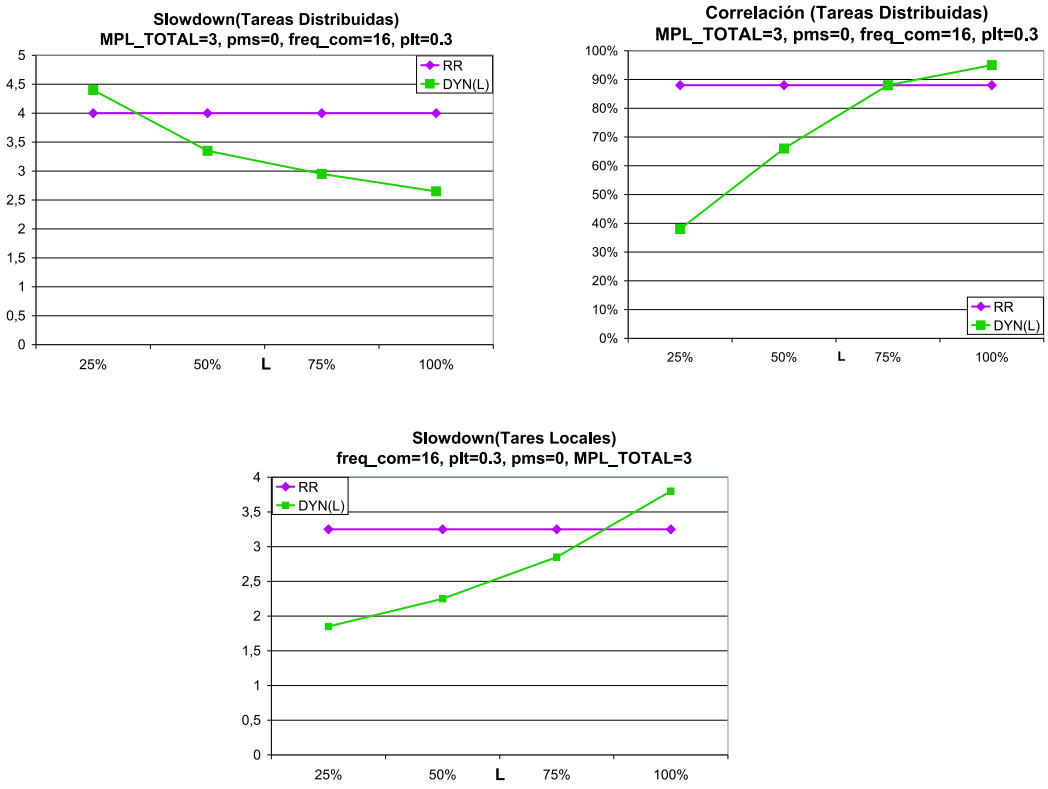

Figura 2.5: Slowdown y correlación de las tareas distribuidas (arriba) y slowdown de las tareas locales (abajo) al variar el porcentaje de recursos asignados a las tareas distribuidas (L).

En general, los resultados obtenidos para las tareas locales reflejan la necesidad de implementar mecanismos adicionales que limiten los recursos de cómputo utilizados por las tareas distribuidas de acuerdo con las necesidades de las tareas locales. Con esta idea se analizó la influencia producida por la limitación en el porcentaje de recursos asignados a las tareas distribuidas,

indicado mediante el acrónimo L, en la ejecución tanto de las tareas locales como de las distribuidas. Con este fin se ha generado una carga caracterizada por un grado de multiprogramación fijo ($MPL\_TOTAL{=}3$) y con una probabilidad de trabajos locales generados constante (plt = 0,3). En este escenario, en aquellos nodos donde existe un usuario local, el quantum asignado a las tareas distribuidas ($task.q$) ha sido reducido proporcionalmente al valor de L, de acuerdo con la siguiente expresión:

$$task.q = DEF\_QUANTUM * L. \tag{2.7}$$

Los resultados obtenidos al simular dicha carga aplicando el coscheduling Dinámico son mostrados en la figura 2.5 bajo el epígrafe *DYN(L)*. Asimismo, a modo de referencia se muestran los resultados obtenidos con el Round-Robin sin ninguna limitación en el uso de los recursos por parte de la tareas distribuidas. El slowdown de las tareas locales mejora a medida que se decrementa el valor de L, de modo que para un $L{<=}75\,\%$ siempre es menor que tres, resultado que refleja la necesidad de limitar los recursos de cómputo asignados a las tareas distribuidas. Sin embargo, esta ganancia es a costa de una importante disminución en el rendimiento de las tareas distribuidas. Este incremento en el slowdown de las tareas distribuidas va asociado con una importante pérdida en la correlación obtenida. Esta baja correlación es debida a que tareas pertenecientes a una misma aplicación distribuida tienen asignadas distintas longitudes de quantum a lo largo del cluster, de modo que mientras a unas tareas se les asigna un quantum igual a $DEF\_QUANTUM$ a otras tareas residentes en nodos con usuarios locales se les asigna un quantum igual a $DEF\_QUANTUM * L$. Estos resultados reflejan la necesidad de implementar un mecanismo para balancear los recursos asignados a las tareas pertenecientes a una misma aplicación, de modo que todas las tareas cooperantes tengan asignado un porcentaje similar de recursos en los nodos donde se ejecuten.

Un problema intrínseco con la técnica del coscheduling Dinámico surge cuando se ejecutan concurrentemente varias copias de una misma aplicación distribuida, de manera que las tareas competidoras son homogéneas y, por tanto, aumenta la probabilidad de que varias tareas en la cola de preparados tengan el mismo número de mensajes en su respectivas colas de mensajes. En esta determinada situación, el planificador no es capaz de discernir entre tareas pertenecientes a diferentes trabajos, lo que comporta que la probabilidad de coscheduling entre tareas cooperantes pueda disminuir drásticamente. Con la idea de validar nuestra intuición, se ha estudiado la sensibilidad de las técnicas de coscheduling Dinámico y del Round-Robin cuando las tareas competidoras tienen la misma frecuencia y patrón de comunicación. Con esta

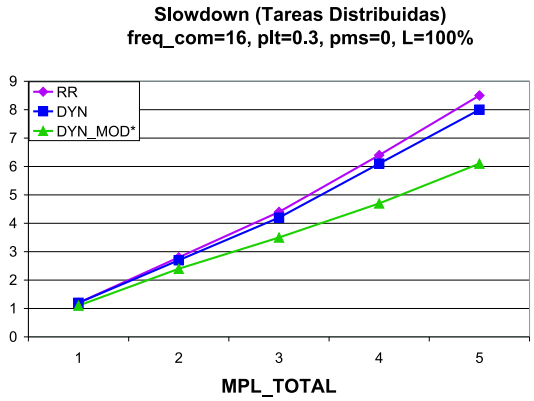

Figura 2.6: Slowdown obtenido al ejecutar MPL_TOTAL trabajos distribuidos iguales.

idea se ha generado una carga distribuida formada por MPL_TOTAL trabajos distribuidos iguales, es decir la similitud entre las tareas es máxima. Los resultados mostrados en la figura 2.6 verifican que el coscheduling Dinámico no es capaz de mantener el coscheduling entre tareas cooperantes con un alto grado de similitud, de manera que prácticamente obtiene unos resultados muy similares a los de la política de Round-Robin. Estos resultados nos motivaron a implementar una modificación del coscheduling Dinámico, de modo que cuando hubiese diferentes tareas competidoras con igual número de paquetes en sus colas de recepción se diese prioridad a aquellas tareas pertenecientes a la aplicación que llevase menos tiempo ejecutándose en el sistema. El hecho de priorizar el trabajo que hace menos tiempo que fue lanzado al sistema, no es una decisión arbitraria, si no que se sustenta en diferentes estudios realizados en entornos multiprogramados dedicados [FR96, Fei97, FRS+97] donde se demuestra la eficacia de las políticas que priorizan aquellas tareas que hace menos tiempo que fueron lanzadas a ejecutar, dado que se considera una aproximación de la política de priorizar la tarea más corta primero (SJB "*Shorter Job First*"). Los resultados mostrados en la figura 2.6 muestran el buen funcionamiento de nuestra propuesta.

Como se explicó en el capítulo de introducción, diferentes autores [YJ01, FPCF01] han puesto de manifiesto la sensibilidad del rendimiento de las técnicas de coscheduling con respecto de la longitud del quantum. Con objeto de corroborar esta hipótesis se simularon varias ejecuciones con diferentes grados de multiprogramación y con diferentes valores de longitud del quantum. En dicha simulación se generó una carga distribuida con una frecuencia de

comunicación alta ($freq\_com = 16$) y con bajos requerimientos de memoria ($pms = 0$).

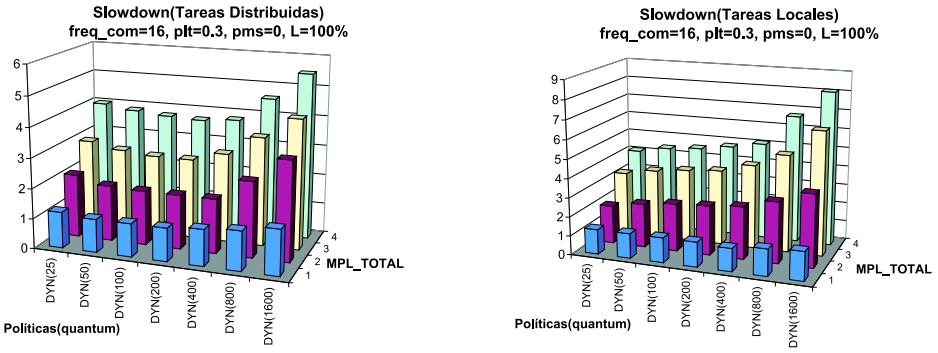

Figura 2.7: Slowdown de las tareas distribuidas (izquierda) y locales (derecha) para distintas longitudes de quantum y grados de multiprogramación.

La figura 2.7(izq.) muestra el slowdown de las tareas distribuidas obtenidos con el coscheduling Dinámico (DYN), al variar las longitudes de quantum en un rango entre [25u.t.,...,1600u.t.][1] y el grado de multiprogramación entre uno y cuatro. Los resultados obtenidos muestran como el slowdown de las tareas distribuidas aumenta a medida que se incrementa el grado de multiprogramación y la longitud del quantum por encima de $400u.t.(= 4xDEF\_QUANTUM)$. Este comportamiento es debido a que el tiempo que las tareas cooperantes deben esperar en la cola de preparados a ser planificadas, debido a que no cumplen la condición de apropiación del coscheduling Dinámico, aumenta a medida que crecen las tareas competidoras así como el quantum de tiempo asignado a cada tarea. Asimismo se observa que un quantum inferior a $50u.t.(= DEF\_QUANTUM/2)$ degrada el rendimiento de las tareas distribuidas, resultado que sugiere que las tareas distribuidas no tienen tiempo suficiente para poder establecer el coscheduling entre ellas.

La figura 2.7(der.) muestra el slowdown obtenido para las tareas locales en las simulaciones anteriormente descritas. Se observa como para un grado de multiprogramación superior a uno, el slowdown aumenta exponencialmente a medida que se incrementa el quantum de tiempo por encima de $200u.t.(= 2xDEF\_QUANTUM)$ alcanzando unos valores de slowdown por encima de dos para un $MPL\_TOTAL > 2$. Estos resultados indican que las máquinas ocupadas por el usuario local deben fijar una longitud del quantum igual o inferior a $200u.t.(= 2xDEF\_QUANTUM)$. Asimismo, para un quantum

[1]La longitud del quantum por defecto (DEF_QUANTUM) es igual a 100u.t.

determinado observamos como el slowdown se incrementa con respecto al grado de multiprogramación, hecho que corrobora la necesidad de limitar los recursos de cómputo utilizados por las tareas distribuidas en entornos con grados de multiprogramación paralela superior a uno.

2.3.2. Rendimiento de las aplicaciones locales y distribuidas en un entorno con altos requerimientos de memoria

Diferentes estudios (ver sección 1.3), reflejan que los requerimientos de memoria es un elemento clave a tener en cuenta en el rendimiento de las tareas distribuidas en entornos NOW dedicados. Sin embargo, en entornos NOW no dedicados, la variabilidad de los requerimientos de memoria por parte del usuario local, así como la interacción entre las necesidades de un usuario local y distribuido, dificultan, en gran medida, las tareas de planificación atendiendo a criterios de memoria. Con objeto de analizar el rendimiento de las tareas distribuidas en entornos con altos requerimientos de memoria, se ha eliminado la restricción, impuesta en las cargas descritas en la sección anterior, de que todas las tareas encajen en memoria. En concreto, se ha generado una carga distribuida formada por tareas con altos requerimientos de memoria, las cuales han sido ejecutadas con diferentes grados de multiprogramación. De este modo, se podrá estudiar el efecto producido por la paginación en la ejecución de las tareas locales y distribuidas.

En la figura 2.8(izq.) se muestra el slowdown, la correlación y los fallos de página obtenidos al variar el grado de multiprogramación entre uno y cinco. El análisis del slowdown muestra como el rendimiento del coscheduling Dinámico disminuye drásticamente al aumentar el grado de multiprogramación por encima de dos, de manera que se alcanzan unos valores de slowdown similares a los obtenidos con la política de Round-Robin original. Un grado de multiprogramación por encima de dos conlleva una elevada carga de memoria ($mem\_load \geq 1{,}3$), la cual provoca una elevada paginación entre la memoria swap y la memoria principal. Esta elevada tasa de paginación provoca que exista una elevada probabilidad de que la tarea receptora de un mensaje no pueda ser planificada debido a que esté en la cola de espera por la ocurrencia de un fallo de página. Este hecho se refleja también en una importante disminución de la métrica de correlación, a medida que se incrementa el grado de multiprogramación por encima de dos y, por consiguiente, los requerimientos de memoria de la carga distribuida. Esta pobre correlación, por debajo del $50\,\%$ en algunos casos, ratifica el hecho de que una paginación excesiva en unos pocos nodos del cluster, pueden provocar

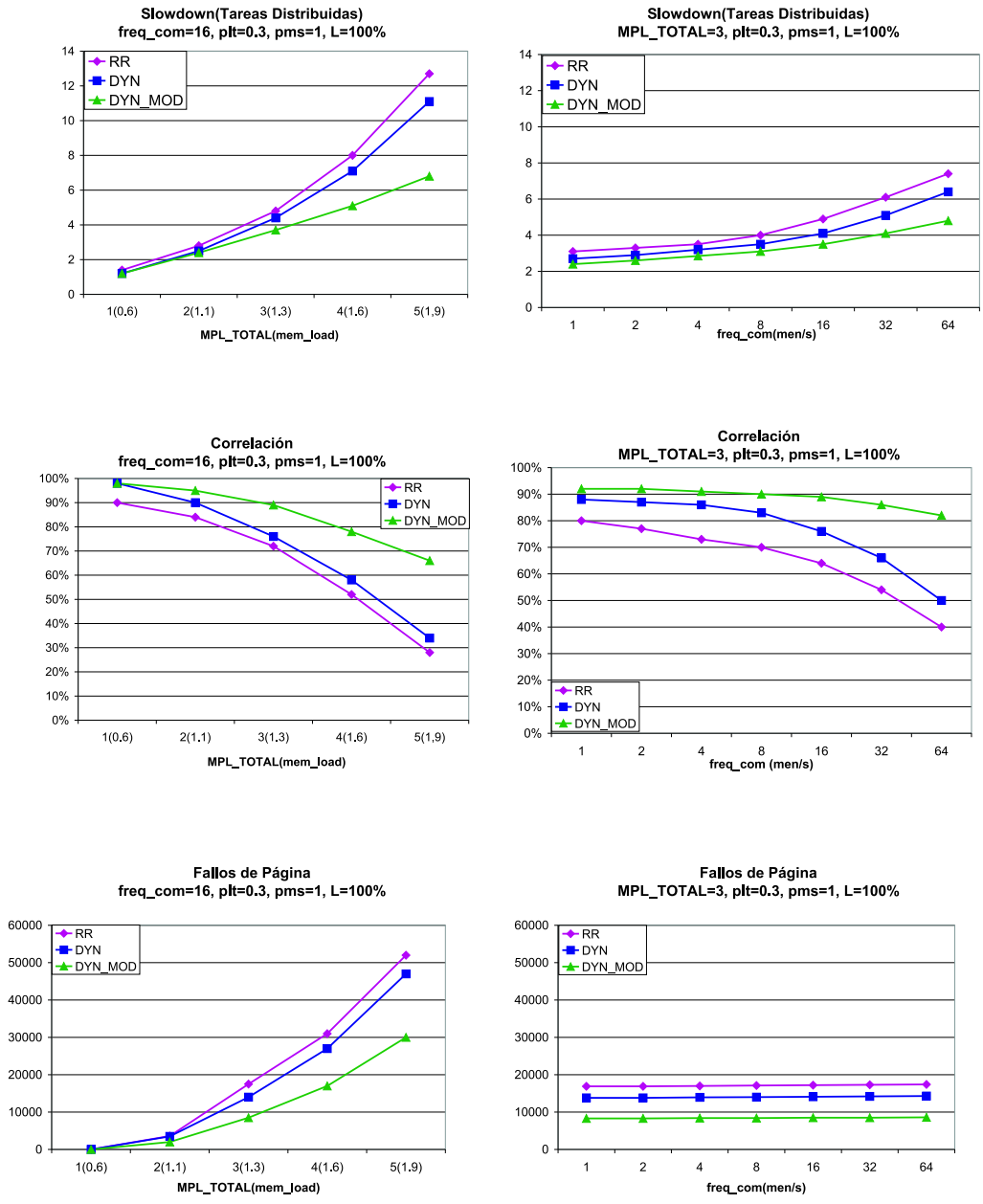

Figura 2.8: Influencia de la paginación en el rendimiento de las tareas distribuidas. En el eje de ordenadas de las figuras de la izquierda se muestra tanto el grado de multiprogramación como la carga media de memoria (entre paréntesis).

una disminución drástica en el rendimiento global de todo el cluster.

El efecto negativo de la paginación se multiplica a medida que aumentan los requerimientos de comunicación de las tareas distribuidos. Este comportamiento se demuestra en la evolución de las figuras 2.8(der.), donde se muestra la variación del slowdown, la correlación y los fallos de página de las tareas distribuidas para distintos ratios de comunicación (*freq_com*) y con un grado de multiprogramación constante igual a 3. Los resultados obtenidos muestran que mientras la tasa de fallos se mantiene prácticamente constante en las tres políticas analizadas, el slowdown se incrementa considerablemente a medida que se aumenta la frecuencia de comunicación. Estos resultados son debidos a que un fallo de página en un nodo puede provocar retrasos en cascada en el resto de nodos donde se ejecutan las tareas cooperantes, debido a la imposibilidad de alcanzar el coscheduling entre dichas tareas, con la consiguiente pérdida de rendimiento que ésto comporta. Asimismo, estos resultados sugieren que aplicaciones distribuidas con granularidad fina (aquellas con elevadas tasas de sincronización) deberían de evitar, en la medida de lo posible, hacer uso de la memoria virtual.

Estos resultados descritos reflejan la necesidad de implementar mecanismos para disminuir esta paginación. Para corroborar este hecho se ha realizado una nueva implementación de coscheduling, indicada como *DYN_MOD*, caracterizada por priorizar la ejecución de aquellas tareas con menor tasa de fallos de página. De esta manera, aquellas tareas que mejor encajan su working set en memoria principal finalizarán antes su ejecución, de modo que, tras su finalización, dejarán disponible una serie de recursos de memoria para el resto de tareas distribuidas. Los resultados mostrados en la figura 2.8 revelan el acierto de nuestra intuición. El número de fallos de página obtenidos con la aplicación de la nueva política DYN_MOD se reduce de una manera importante, hecho que repercute en una mejora considerable, tanto de la correlación como del slowdown de las tareas distribuidas.

En estas condiciones de alta demanda de memoria parece lógico pensar que el hecho de ajustar la longitud del quantum a los requerimientos de memoria de los procesos puede repercutir en una mejora del rendimiento del subsistema de memoria. Esto es debido a que el working set de las aplicaciones distribuidas consiste en una serie de fases durante las cuales predomina una elevada localidad [BHMW96], de modo que al finalizar cada fase es cuando se producen la gran mayoría de fallos de página debido a la carga del conjunto de páginas que serán referenciadas durante la posterior fase. Este comportamiento sugiere que si se ajusta la longitud del quantum a la duración de cada fase, el número de fallos de paginas se debería reducir, y por consiguientes el slowdown de las tareas distribuidas. Este comportamiento se refleja perfectamente en las gráficas mostradas en la figura 2.9, donde el slowdown

y los fallos de página disminuyen rápidamente hasta alcanzar una longitud de quantum de $400u.t.$, de modo que a partir de este valor la tendencia se suaviza.

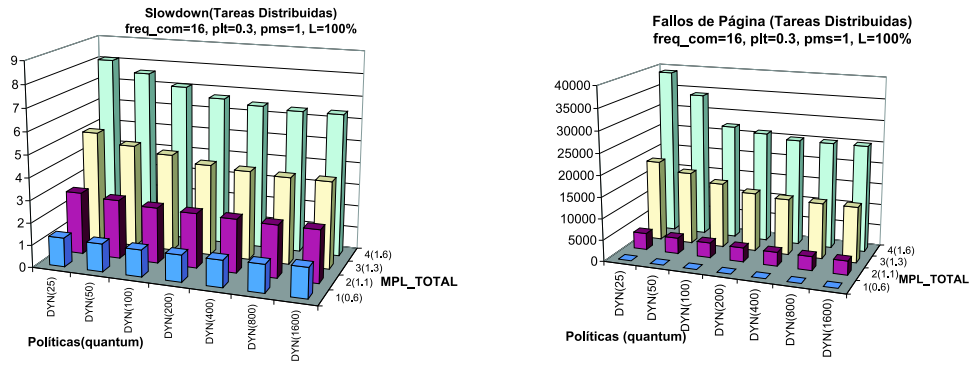

Figura 2.9: Evolución del slowdown y fallos de página de las tareas distribuidas con altos requerimientos de memoria al variar las longitudes de quantum.

La figura 2.10 muestra el slowdown y overhead de las tareas locales al ser ejecutadas junto con trabajos distribuidos con altos requerimientos de memoria. Los mejores resultados son obtenidos con la nueva técnica sugerida (DYN_MOD) debido a que esta técnica disminuye el número de fallos de página para todas las tareas, tanto locales como distribuidas, hecho que repercute en mejorar considerablemente el slowdown. Sin embargo, para un grado de multiprogramación superior a uno, el slowdown obtenido es superior a dos, resultado que puede ser considerado inaceptable para la realización de determinadas tareas de cómputo intensivo asociadas a un usuario local, como por ejemplo, la compilación de un núcleo de un sistema operativo. Estos malos resultados son debidos a que las políticas evaluadas no discriminan a la hora de seleccionar la tarea para intercambiar páginas, con lo que las tareas locales sufren directamente el efecto de la paginación producido por los elevados requerimientos de las tareas distribuidas. Estos resultados sugieren la necesidad de dividir la memoria en dos áreas distintas, un porcentaje de memoria para las tareas locales y el resto para las distribuidas. De esta manera, si el desbordamiento de memoria es debido a que las tarea distribuidas sobrepasan la porción de memoria asignada a las mismas, como es el caso estudiado, solamente las tareas distribuidas se vean penalizadas por dicho intercambio. Asimismo, estos resultados reflejan la necesidad de que el grado de multiprogramación paralelo varíe dinámicamente en función de los recursos de memoria disponibles.

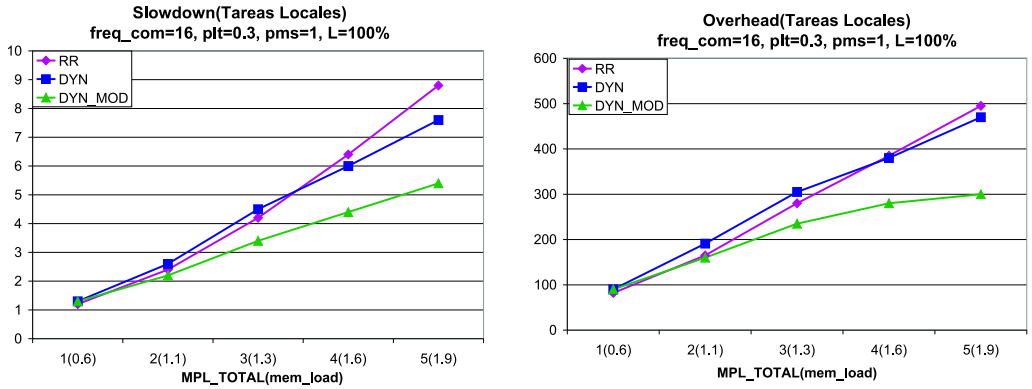

Figura 2.10: Slowdown y overhead de las tareas locales al ser ejecutadas con una carga distribuida con altos requerimientos de memoria.

2.3.3. Influencia de la memoria cache y la longitud del quantum en el rendimiento de las tareas distribuidas

Diferentes estudios realizados en entornos *NOW dedicados* [WMADC99] han puesto de manifiesto la correlación existente entre el rendimiento de las aplicaciones distribuidas y el rendimiento de la memoria cache, siendo este último muy dependiente de la longitud del quantum [SRD01]. Este comportamiento es debido a que longitudes de quantum excesivamente cortas pueden degradar el rendimiento de la memoria cache, dado que los procesos no tienen tiempo de explotar toda la localidad asociada con su working set cargado en la memoria cache tras la reanudación de su ejecución después de un cambio de contexto. Por este motivo, desde el punto de vista del rendimiento de la cache, la longitud óptima del quantum depende de las características de memoria de los trabajos concurrentes, de manera que un quantum dado puede ser suficiente para procesos con pequeños working set pero en cambio puede quedarse corto para procesos con working sets grandes. Con el objetivo de analizar esta correspondencia se procedió a estudiar el rendimiento de la cache con respecto de la longitud del quantum, así como su influencia en el rendimiento de las tareas distribuidas en un cluster real controlado.

2.3.3.1. Entorno experimental utilizado

Nuestro estudio se ha realizado en un cluster formado por 8 Pentium III, conectados mediante una red Fast Ethernet, que tienen las siguientes características:

- Frecuencia de reloj de 801.42Mhz.

- Cache L1 de 32KB y cache L2 de 256 KB (ambas integradas en la CPU).
- Memoria RAM de 256 MB.
- Memoria Swap de 256MB.
- Tarjeta de red Fast Ethernet 3Com 3c905C-TX.
- Sistema Operativo Linux red Hat 6.2. (kernel v.2.2.15).

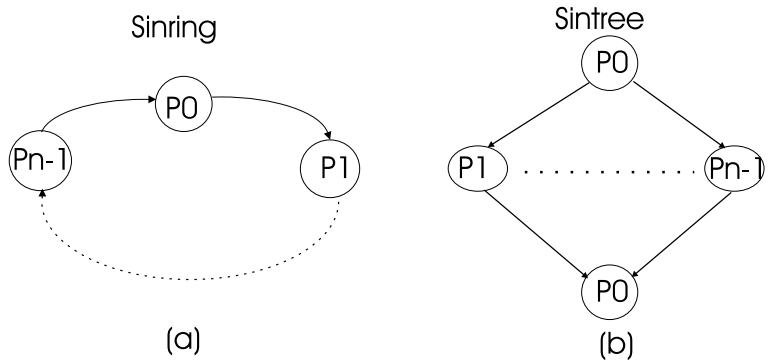

Figura 2.11: Patrón de comunicación de Sintree y Sinring.

Con objeto de controlar los requerimientos de la carga distribuida ejecutada, dos diferentes benchmarks sintéticos, denominados *Sinring* y *Sintree* respectivamente, han sido utilizados. Cada uno de estos benchmarks realiza reiteradamente el mismo proceso: Sinring implementa un anillo lógico (ver figura 2.11(a)), mientras que Sintree se caracteriza por una comunicación de uno a varios y de varios a uno (ver figura 2.11(b)). En ambos benchmarks, cada una de las tareas que constituye la aplicación realiza, ante la recepción de un mensaje, un cómputo determinado durante un tiempo fijado por el usuario; siendo este parámetro pasado como argumento ($Tcomp$). Una vez, cada tarea ha finalizado dicho cómputo envía un nuevo mensaje, de tamaño fijado por el usuario ($size$), al proceso cooperante correspondiente. De este modo, la frecuencia de envío de mensajes ($freq$) de cada tarea vendrá fijada por los parámetros $Tcomp$ y $size$. Asimismo, ambos benchmarks permiten al usuario de los mismos fijar el número de tareas distribuidas que forman la aplicación distribuida (N), el tamaño de memoria utilizada por cada una de las tareas (mem) y el número de iteraciones a realizar ($iter$). La tabla 2.1 muestra los parámetros utilizados, por defecto, en la experimentación realizada, así como los tiempos de ejecución obtenidos con ambos benchmarks

en el cluster totalmente dedicado (MPL=1 y sin usuarios locales). Una extensión del estudio descrito en esta sección, puede ser consultada en [Bai02]. En este trabajo se muestran los resultados obtenidos con diferentes cargas distribuidas, formadas por un conjunto de benchmarks pertenecientes a la NAS suite [BBB+94].

	N	*Tcomp*	*mem*	*size*	*iter*	*t.ejec.*
Sinring	8	200msg	256KB	1KB	100	112s
Sintree	8	200msg	256KB	1KB	100	64s

Tabla 2.1: Tiempos de ejecución de Sintree y Sinring en un entorno dedicado.

Las tareas locales han sido simuladas mediante un benchmark sintético, denominado *Local*, que permite fijar los requerimientos de memoria, CPU, E/S a disco y comunicación utilizados por un usuario local. Asimismo, con objeto de conocer el grado de intrusión introducido por la ejecución de las aplicaciones distribuidas en las tareas locales, este mismo benchmark retorna el tiempo de ejecución en realizar un determinado cómputo junto con la latencia media en realizar un determinado número de llamadas a sistema. El apéndice E describe en detalle la implementación realizada de este benchmark, así como el grado de fiabilidad del mismo con respecto al comportamiento de un usuario local real.

Esta pruebas fueron realizadas en un cluster real, dada la facilidad con la que se pueden monitorizar los eventos de cache en los procesadores actuales. La familia de procesadores P6 disponen de un conjunto de registros, denominados *MSRs* ("*Model Specific Registers*"), que permiten monitorizar la ocurrencia de determinados eventos en el procesador, como por ejemplo los fallos de cache [Int02a, Int02b, Int02c]. A partir de la monitorización de dichos contadores se calculó la tasa de fallos de la cache (mr), de acuerdo con la siguiente expresión:

$$mr = \sum_{j=1}^{n} \frac{\sum_{i=1}^{T} \frac{node_j.L2\_LINES\_IN_i}{node_j.L2\_RQSTS_i}}{n * T} * 100, \tag{2.8}$$

donde $node_j.L2\_LINES\_IN_i$ y $node_j.L2\_RQTS_i$ son el número total de fallos de la cache L2 y el número total de peticiones de la cache L2 (contabilizando tanto las del kernel como las del usuario) producidos durante el i-ésimo segundo en el nodo j, respectivamente; T es el número de muestras capturadas durante el intervalo de monitorización y n es el número de nodos del cluster. La programación y lectura de dichos contadores fue realizada me-

diante el uso de un driver específico, denominado *perfctr* [Pet02], desarrollado por la *universidad de Tennessee* para este propósito.

Asimismo, con objeto de modificar el quantum del sistema operativo, se implementó una nueva llamada a sistema que permite multiplicar el quantum base que utiliza el s.o. Linux por una variable, denominada $STEP$, la cual es pasado como argumento de dicha llamada. De este modo, el quantum de planificación de Linux es modificado de acuerdo con la siguiente expresión:

$$quantum = DEF\_QUANTUM * STEP, \tag{2.9}$$

donde $DEF\_QUANTUM$ es el quantum base utilizado por el sistema operativo Linux ($= 210ms$). En el capítulo 4 de implementación se realiza una descripción detallada del mecanismo utilizado por Linux para asignar a cada tarea su correspondiente quantum de planificación.

Finalmente, en el kernel de cada uno de los nodos del cluster se instaló un parche con la implementación asociada al coscheduling Dinámico. Esta implementación del coscheduling Dinámico es totalmente equivalente a la descrita anteriormente en la sección de simulación. Una explicación detallada de dicha implementación puede encontrarse en [Sol02].

2.3.3.2. Análisis de los resultados experimentales obtenidos

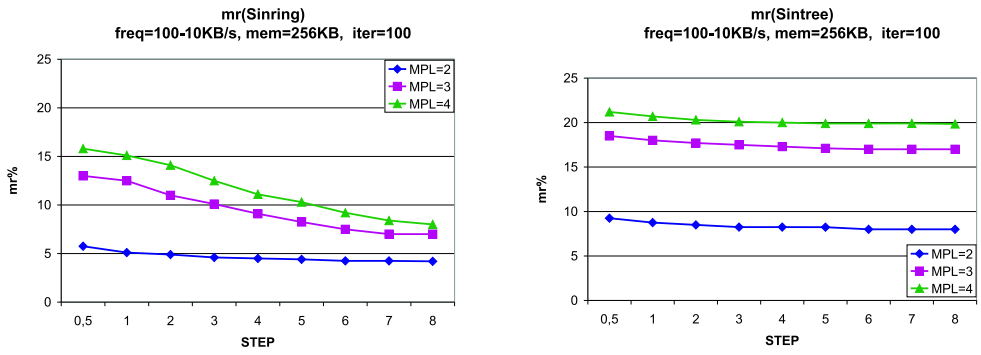

Figura 2.12: Rendimiento de la cache con respecto de la longitud del quantum.

Con objeto de evaluar la influencia de la cache en la ejecución de aplicaciones distribuidas con diferentes patrones de comunicación, se generaron dos cargas distribuidas diferentes. Cada una de ellas ejecuta concurrentemente MPL aplicaciones, Sinring o Sintree, respectivamente. La figura 2.12 muestra el ratio de fallos de la cache (mr) obtenidos para cada una de las

cargas generadas, al variar el quantum entre 100ms ($STEP = 0{,}5$) y 1,6s ($STEP = 8$) y el grado de multiprogramación (MPL) entre 2 y 4. Con objeto de que no se ejecutasen concurrentemente aplicaciones con un mismo ratio de comunicación, de modo que el rendimiento del coscheduling dinámico se pudiera ver perturbado, las diferentes aplicaciones distribuidas que forman parte de una misma carga fueron generadas con frecuencias de comunicación distintas, comprendidas en el intervalo mostrado en el título del gráfico ($freq = [100KB/s, ..., 10KB/s]$). El análisis de ambas gráficas muestra como el ratio de fallos de cache aumenta con el grado de multiprogramación y disminuye a medida que se incrementa la longitud del quantum. La comparación entre ambas gráficas revela como esta disminución con respecto de la longitud del quantum depende del patrón de comunicación de las aplicaciones; mientras la carga formada por aplicaciones Sinring obtiene una disminución alrededor del *8 %*, en la carga formada por aplicaciones Sintree esta disminución no supera el *2 %* en el mejor de los casos (MPL=4).

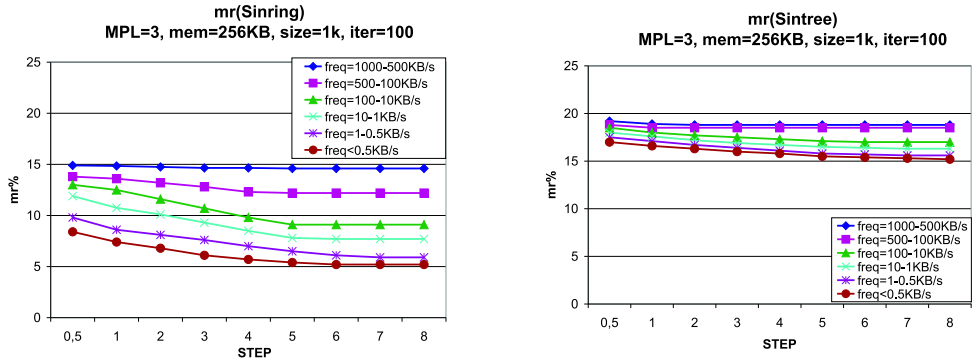

Figura 2.13: Rendimiento de la cache con respecto la frecuencia de comunicación (freq).

Con la idea de comprobar esta dependencia del rendimiento de la cache con respecto a los requerimientos de comunicación de las tareas distribuidas, se generaron un conjunto de nuevas cargas, para ambos benchmarks, con un grado de multiprogramación fijo ($MPL = 3$) y con ratios de comunicación comprendidos en intervalos con diferentes órdenes de magnitud, de modo que el más grande corresponde al intervalo $[1000KB/s, ..., 500KB/s]$ y el más pequeño al intervalo $[0{,}5KB, ..., 0{,}1KB/s]$. Los resultados mostrados en la figura 2.13 reflejan, para ambas cargas, como para aplicaciones con altas frecuencias de comunicación ($freq \geq 100KB/s$), el ratio de fallos de cache se mantiene prácticamente constante, mientras que aplicaciones con bajos ratios de comunicación son más sensibles a la longitud del quantum, de modo que

el ratio de fallos de cache disminuye a medida que se aumenta el $STEP$. Este comportamiento se explica teniendo en cuenta la propia naturaleza del coscheduling Dinámico. Dado que el coscheduling Dinámico planifica las tareas distribuidas de acuerdo con el número de paquetes de comunicación recibidos por las mismas, las tareas con elevadas frecuencias de comunicación serán replanificadas constantemente. Este elevado trasiego provoca la expropiación de la CPU de las tareas antes de que hayan consumido su quantum asignado y, en consecuencia, no hayan podido explotar, en la mayoría de los casos, toda su localidad. Estos resultados implican que el rendimiento de la cache, para aplicaciones con elevados ratios de comunicación, es independiente de la longitud del quantum.

La figura 2.14 muestra los tiempos de ejecución medios obtenidos, para las mismas cargas distribuidas, al variar la longitud del quantum y el grado de multiprogramación. Los resultados obtenidos para la carga formada por las aplicaciones Sinring (ver figura 2.14(izq.)) muestran una ligera ganancia (alrededor del 3 %) respecto del valor obtenido con el quantum original ($STEP = 1$) a medida que se incrementa la longitud del quantum hasta un límite de $800ms$ ($STEP = 4$). Esta ganancia es debida principalmente a la mejora en el rendimiento de la cache para estos valores de quantum (ver figura 2.12). Sin embargo, a partir de este umbral, el tiempo de ejecución empieza a empeorar como consecuencia de la pérdida de coscheduling con longitudes de quantums grandes, efecto descrito en el apartado anterior de simulación (ver sección 2.3.1). Esta perdida de coscheduling se ve amplificada en los resultados obtenidos por la aplicación Sintree (ver figura 2.14), debido a que esta aplicación comunica más y, en consecuencia, es más sensible al efecto del coscheduling.

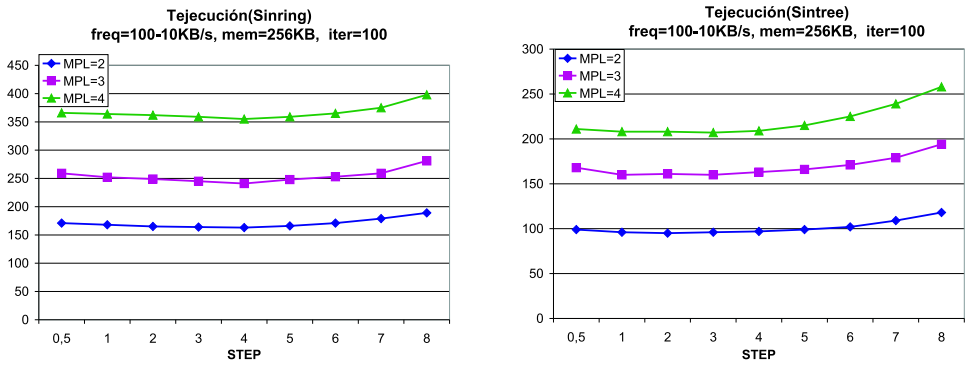

Figura 2.14: Tiempo de ejecución de Sinring y Sintree para diferentes grados de multiprogramación y diferentes longitudes de quantum.

Finalmente, se repitieron las pruebas anteriores añadiendo al sistema cua-

tro usuarios locales. Los argumentos del benchmark Local utilizado para simular los requerimientos de cómputo de los usuarios locales fueron los siguientes:

- Carga CPU=0.15.
- Memoria=35 % del tamaño de la memoria principal.
- E/S a disco (bloques/s leidos-escritos)= 67 - 17.
- Red (paquetes/s recibidos - enviados) = 608 - 30.

Estos parámetros del benchmark Local se corresponden con el perfil *Xwin*, descrito en el apéndice E. Asimismo, teniendo en cuenta los resultados anteriormente obtenidos en el entorno simulado, en aquellos nodos con usuario local se mantuvo fijo el quantum por defecto asignado por Linux, mientras que en el resto de nodos éste fue modificado de acuerdo con los valores del STEP. Los resultados obtenidos, mostrados en la figura 2.15, reflejan que la ganancia anteriormente obtenida en el entorno dedicado debida a la mejora en el rendimiento de la cache para valores de STEP alrededor de cuatro, se ve diluida debido al efecto de la descoordinación introducida en la asignación de las longitudes del quantum a lo largo del cluster.

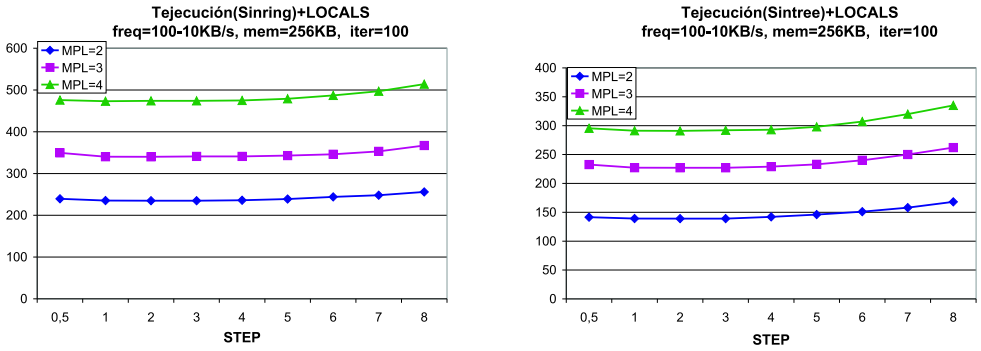

Figura 2.15: Tiempo de ejecución de Sinring y Sintree en un entorno no dedicado para diferentes grados de multiprogramación y diferentes longitudes de quantum.

A partir de los resultados analizados, podemos concluir que en aquellos nodos en los cuales no existen usuarios locales, el rendimiento de la cache aumenta progresivamente a medida que se incrementa la longitud del quantum hasta alcanzar un límite máximo de $800ms$. Esta ganancia repercute en una ligera mejora en los tiempos de ejecución de las correspondientes aplicaciones

distribuidas. A partir de dicho límite, el tiempo de ejecución empieza a disminuir debido a la pérdida de coscheduling. Estos resultados muestran que esta ganancia es muy susceptible del patrón y frecuencia de comunicación de las correspondientes aplicaciones distribuidas, de modo que disminuye a medida que aumentan las frecuencias de comunicación de las tareas distribuidas. Asimismo, esta ganancia desaparece en aquellos entornos en los cuales existe algún usuario local.

Finalmente, cabe decir que los resultados descritos en [Bai02] con una carga formada por distintos benchmarks distribuidos de la NAS suite son muy similares a los mostrados a lo largo de esta sección.

Capítulo 3

CoScheduling Cooperativo (CSC)

Los resultados obtenidos en el capítulo anterior muestran que las técnicas tradicionales de coscheduling con control implícito, basadas exclusivamente en la ocurrencia de determinados eventos de comunicación, no son capaces de adaptarse a la alta variabilidad que caracteriza a los entornos cluster no dedicados multiprogramados. Estos resultados ponen de manifiesto la necesidad de desarrollar nuevas técnicas de coscheduling que abarquen, en la medida de lo posible, todos aquellos parámetros que influyen en la coordinación de las aplicaciones distribuidas en un cluster no dedicado. Con este fin, en este capítulo se presenta una nueva propuesta de coscheduling, denominada CoScheduling Cooperativo (CSC), la cual gestiona los recursos de CPU, a nivel de longitud de quantum y de prioridad de planificación, y de memoria a lo largo del cluster. Para ello, CSC tiene en cuenta la ocurrencia tanto de diferentes eventos locales, como son la recepción y emisión de mensajes, los fallos de página y la actividad de un usuario local, como de eventos remotos, como son la recepción de información de sistema procedente de nodos remotos, asociada a la actividad de usuario y de memoria.

En este capítulo, en primer lugar se describe, a nivel de diagrama de bloques, las diferentes partes que configuran esta nueva propuesta de coscheduling, así como la interacción entre las mismas y con los diferentes subsistemas que gestionan. Posteriormente, se procede a explicar, a nivel algorítmico, cada uno de estos módulos, con especial énfasis en el modo utilizado para gestionar los recursos y asignarlos a partir de la ocurrencia de unos determinados eventos. Finalmente, por medio de la simulación, se muestra como CSC aumenta el rendimiento de las técnicas de coscheduling tradicionales debido a su capacidad de adaptarse dinámicamente a la variabilidad de los entornos cluster no dedicados, explotando eficientemente los recursos de cómputo disponibles en los mismos. De este modo, la simulación de CSC permite validar la viabilidad de su implementación, así como determinar las cotas

máximas de rendimiento de CSC.

3.1. Elementos que Configuran el CoScheduling Cooperativo

De acuerdo con los resultados obtenidos en el capítulo anterior, un rendimiento eficiente, tanto de las aplicaciones distribuidas como de las locales en un entorno cluster no dedicado conlleva una gestión coordinada, a lo largo de todo el cluster, tanto del subsistema de planificación de la CPU, a nivel de la longitud de quantum y de la prioridad de planificación, como del subsistema de memoria, de modo que se adapte a las necesidades de ambos tipos de aplicaciones.

Con este fin, CSC está constituido por diferentes módulos, cada uno de los cuales se encarga, por separado, de la gestión, a nivel local, de uno de los subsistemas mencionados o bien del intercambio de información de sistema con el resto de nodos cooperantes. La interacción entre todos estos módulos permite a CSC gestionar eficientemente todos los recursos de cómputo a lo largo del cluster, con el objetivo de ejecutar múltiples aplicaciones distribuidas, de un modo coordinado, sin perjudicar el trabajo del usuario local.

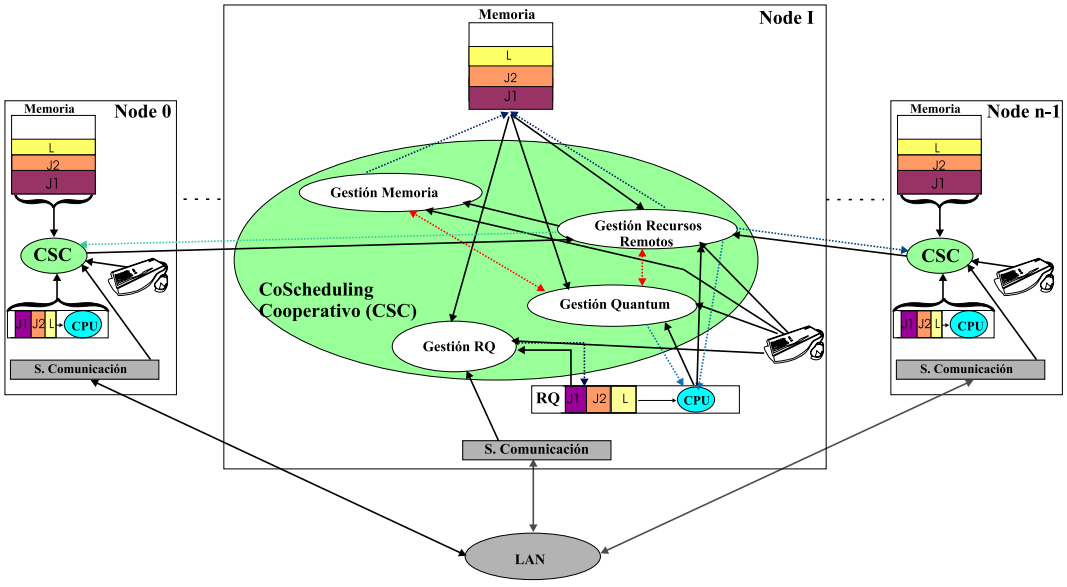

Figura 3.1: Diagrama de bloques del CoScheduling Cooperativo.

CSC, siguiendo el principio de coscheduling con control implícito, es un

sistema totalmente distribuido (ver figura 3.1), de modo que los recursos de cómputo de cada nodo son gestionados por el coscheduling cooperativo residente en el mismo. La figura 3.1 muestra los diferentes elementos que constituyen CSC, así como los subsistemas que gestionan (flechas azules discontinuas), los eventos que reciben y analizan (flechas negras continuas) y las diferentes relaciones entre los mismos (flechas rojas discontinuas). En concreto, CSC está formado en cada nodo por los siguientes cuatro módulos:

- **Módulo de Gestión de Memoria.** Los resultados obtenidos en el capítulo anterior ponen de manifiesto la necesidad de tener en cuenta los requerimientos de memoria en la planificación de los trabajos paralelos en un entorno cluster multiprogramado. En primer lugar es necesario asegurar, tanto a las tareas locales como a las distribuidas, un porcentaje de memoria suficiente para que éstas puedan progresar, de modo que los usuarios, tanto locales como de las aplicaciones distribuidas, conozcan a priori los recursos de memoria disponibles y en función de los mismos puedan ejecutar sus correspondientes cargas. Asimismo, un uso eficiente de un cluster no dedicado por parte de las tareas distribuidas con altos requerimientos de comunicación conlleva minimizar el efecto negativo producido por una excesiva paginación en el rendimiento de dichas tareas paralelas. Con este objetivo, este módulo se encarga de reducir dinámicamente el grado de multiprogramación de las tareas distribuidas, parando y desalojando de memoria aquella tarea distribuida con un mayor requerimiento de memoria. De acuerdo con este doble propósito, el bloque de gestión de memoria se encarga tanto de dividir como de gestionar la memoria en dos diferentes porciones, una para el usuario local y la otra para el usuario de las aplicaciones paralelas; garantizando para ambos usuarios su correspondiente porción, incluso en el caso de que se produzca un desbordamiento de memoria debido a un exceso de requerimientos, por parte de uno o bien de ambos usuarios.

- **Módulo de Gestión de la Cola de Preparados (RQ)**. Este módulo se encarga de ordenar las tareas en la cola de preparados de acuerdo con sus necesidades de comunicación o de memoria. En caso de que la memoria principal no esté desbordada, las tareas son ordenadas en función del número de mensajes recibidos/enviados, con el objetivo de minimizar el tiempo de espera de las tareas cooperantes; mientras que en caso de que la memoria sea desbordada, las tareas se ordenan en función de su correspondiente tasa de fallos de página, con el objetivo de minimizar la paginación.

- **Módulo de Gestión de la Longitud del Quantum**. Al igual que en el caso anterior, este bloque tiene un doble comportamiento, atendiendo al estado de la memoria. De este modo, en caso de que la memoria no sea desbordada, la longitud del quantum es asignada de acuerdo con la presencia del usuario local, con el objetivo de fijar el máximo retardo introducido en el trabajo del usuario local, así como en función del estado global de todas las tareas que forman una misma aplicación distribuida. En el caso de que la memoria sea desbordada, la longitud del quantum se incrementará en aquellos nodos donde no haya usuarios locales, con el objetivo de explotar la localidad de las tareas distribuidas y de este modo disminuir el número de fallos de página.

- **Módulo de Gestión de Recursos Remotos**. Este módulo permite balancear los recursos de cómputo a lo largo del cluster, de manera que todas las tareas que pertenecen a una misma aplicación distribuida tengan asignada una misma prioridad con respecto al uso de los recursos de cómputo asociados al nodo donde se ejecutan y, de este modo, puedan progresar de un modo coordinado a lo largo del cluster. Con este fin, este módulo se encarga de enviar, al resto de nodos cooperantes, y recibir, de dichos nodos, todos aquellos eventos que han provocado cambios en la asignación de recursos en las tareas distribuidas. Este hecho provoca que este módulo tenga una elevada interacción con el resto de módulos que constituyen CSC.

En las siguientes secciones se procederá a desglosar el funcionamiento de cada uno de los módulos enumerados, explicando su contribución en el proceso de coscheduling, así como las diferentes interrelaciones entre los mismos.

3.2. CoScheduling Cooperativo: Asignación de la Memoria

En el estudio realizado en el capítulo precedente se puso de manifiesto que una política de asignación de memoria basada en la aplicación de criterios equitativos entre las diferentes tareas de usuario, gestionadas por un s.o. de tiempo compartido, no son compatibles con los objetivos planteados en un cluster no dedicado multiprogramado.

La gestión de memoria, por parte de CSC, se rige mediante un *contrato social*, entre el usuario local y el usuario paralelo, de manera que ambas partes

pactan un porcentaje de recursos de cómputo (L[1]) a utilizar por cada una de ellas. Este pacto significa, con respecto a la memoria, que CSC divide la memoria principal de cada *node* ($node.M$) en dos porciones: una asociada a las tareas locales ($node.M^L$) y otra a las tareas paralelas ($node.M^D$). De este modo, la porción de memoria asociada a las tareas paralelas será calculada de acuerdo con la siguiente expresión:

$$node.M^D = node.M * L, \tag{3.1}$$

donde L es el porcentaje de memoria asignado a las tareas paralelas. El resto de la memoria principal será asignado a las tareas locales ($node.M^L$). Cabe decir que este porcentaje solamente se tendrá en cuenta en aquellas situaciones en las que se desborde la memoria principal, es decir, cuando $node.mem > node.M$. De esta manera, CSC asegura que las tareas paralelas siempre dispongan de un mínimo tamaño de memoria para su ejecución.

En el caso de que la memoria sea desbordada, un algoritmo de reemplazo de páginas global no se corresponde con la filosofía del contrato social descrita anteriormente, dado que un exceso de requerimientos de memoria, por parte de las aplicaciones distribuidas, podría perjudicar el rendimiento de las tareas locales o viceversa. La figura 3.2 ilustra el algoritmo seguido por CSC para seleccionar el proceso sobre el cual se aplicará la paginación; este proceso será indicado como *swaptask*. El modo de proceder del algoritmo para la selección de *swaptask* dependerá de quién rompa el contrato realizado. En función del mismo, se pueden dar dos situaciones diferentes:

- En el caso de que las tareas paralelas superen su porción de memoria asignada ($node.mem\_par > node.M^D$), dos situaciones distintas son contempladas:

  - En color azul se ilustra el caso de que exista, en dicho nodo, alguna tarea notificada con el estado *STOP* o bien exista una tarea previamente detenida (*stoptask*). Una tarea distribuida es identificada como *stoptask* si existe actividad de usuario local ($node.LOCAL\_USER$) en el *node* y además, ya hubiese sido seleccionada previamente como *swaptask*, como ilustra, en color verde, el algoritmo de la figura 3.2. Asimismo, una tarea es notificada como STOP en el caso de que tenga una tarea cooperante que haya sido detenida previamente en un nodo remoto. Esta notificación de eventos ocurridos en nodos remotos será explicado con

[1]El prefijo *node* se omite para enfatizar que el valor de este parámetro es uniforme a lo largo del cluster.

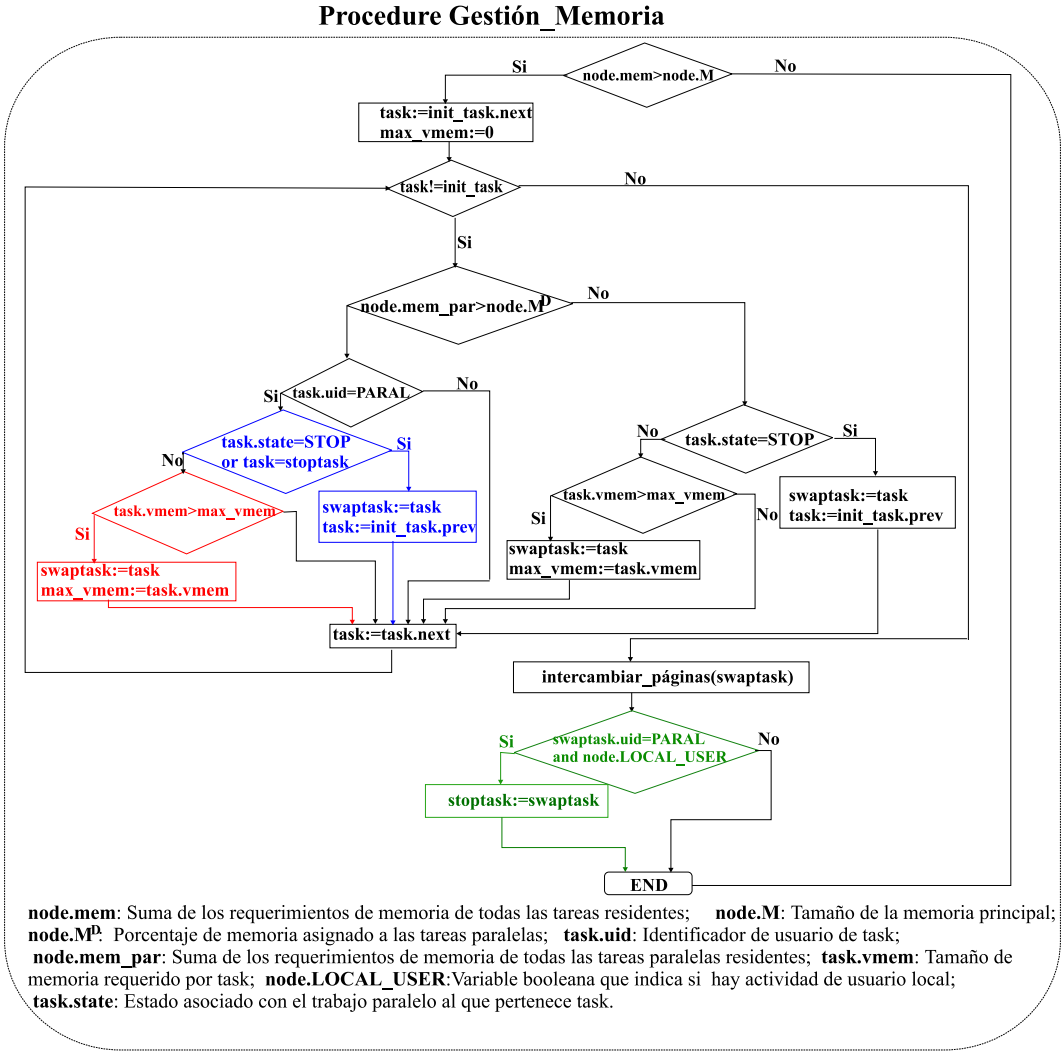

Figura 3.2: Algoritmo de gestión de memoria.

detalle en la sección 3.5. Por tanto, en caso de que exista una tarea notificada como STOP, o bien ya hubiese una tarea identificada como *stoptask*, ésta será escogida para el intercambio de páginas (*swaptask*).

- En caso contrario, indicado en color rojo, el algoritmo seleccionará aquel proceso distribuido con un mayor requerimiento de memoria ($task.vmem > max\_vmem$). Dado que las tareas con mayores requerimientos de memoria tienen asociada, normalmente, una mayor tasa de fallos de página, este criterio se corresponde con los

resultados obtenidos en la simulación, los cuales mostraban que una política basada en priorizar aquellos procesos con menor tasa de fallos de página ayudaba a minimizar el número de fallos de página.

- Por otro lado, en caso de que la memoria sea desbordada debido a que los requerimientos de memoria de las tareas locales superen la porción asignada a las mismas, CSC seleccionará páginas del proceso local o paralelo con un mayor requerimiento de memoria.

Cabe destacar que el funcionamiento descrito, prioriza claramente al usuario local con respecto al paralelo. Sin embargo, esta priorización es coherente con los objetivos de CSC de priorizar el rendimiento de las tareas locales con respecto de las paralelas.

3.2.1. Heterogeneidad de la memoria

En la práctica, un elevado número de clusters se caracterizan por la heterogeneidad respecto de sus recursos de cómputo. En muchos casos, clusters con un elevado número de nodos, se suelen adquirir en sucesivas etapas. Teniendo en cuenta el ritmo al que varía la tecnología, las prestaciones de las nuevas máquinas difieren substancialmente respecto de las antiguas. Esta heterogeneidad afecta, entre otros muchos subsistemas, al tamaño de la memoria principal. De acuerdo con los objetivos de nuestro trabajo, este hecho plantea un problema añadido a la hora de uniformizar los recursos de cómputo asignados a las tareas paralelas que forman parte de un mismo trabajo.

De acuerdo con la política de contrato social descrita anteriormente, cada nodo del cluster asigna una porción de memoria principal a las tareas locales ($node.M^L$) y otra a las tareas paralelas ($node.M^D$), la cual, de acuerdo con la expresión 3.1 explicada en la sección anterior, es proporcional al tamaño de la memoria principal ($node.M$) y al porcentaje de recursos asignados a las tareas paralelas (L). Cabe recordar que este porcentaje es uniforme a lo largo del cluster. Este modo de asignación comporta una ausencia de uniformidad con respecto al tamaño de memoria asignado para las tareas paralelas en aquellos clusters compuestos por nodos con diferentes tamaños de memoria. Por ejemplo, dado un cluster formado por máquinas con tamaño de memoria de 256MB y de 128MB, una reserva de un 25 % de recursos de memoria para las tareas paralelas implica que las tareas paralelas dispondrán de 32MB en aquellos nodos con 128MB y de 64MB en el resto de nodos con un tamaño de memoria igual a 256MB, hecho que dificultará una asignación uniforme

de recursos y, por tanto, que todas las tareas cooperantes puedan progresar uniformemente a lo largo del cluster. Con objeto de paliar este problema, la cantidad de memoria asignada a las tareas paralelas, en entornos heterogéneos, será calculada de acuerdo con la siguiente ecuación:

$$node.M^D = min_{i=1}^{n}(node_i.M) * L, \tag{3.2}$$

donde n es el número de nodos del cluster. De este modo, el mismo tamaño mínimo de memoria será asignado a las tareas paralelas en cada uno de los nodos del cluster, independientemente del tamaño de su memoria principal. Es importante remarcar que en el caso de que las tareas locales no utilicen todos los recursos de memoria que les han sido asignados:

$$node.M^L = node.M - node.M^D, \tag{3.3}$$

las tareas paralelas podrán hacer uso de los mismos. Por tanto, la porción de memoria asignada a las tareas paralelas ($node.M^D$) debe ser considerada como el mínimo tamaño de memoria disponible para la ejecución de las mismas.

Asimismo, esta nueva reasignación de la memoria comporta que el tamaño de memoria disponible para la ejecución de las tareas locales ($node.M^L$) siempre sea igual o superior a lo que el usuario local espera ($node.M^L_{espera}$):

$$node.M^L_{espera} = node_k.M * L, \tag{3.4}$$

manteniendo, de este modo, el objetivo de priorizar la asignación de recursos al usuario local.

3.3. CoScheduling Cooperativo: Asignación de Prioridades

De acuerdo con el modelo de cluster descrito en la sección 2.1.1, la prioridad de planificación de la CPU depende del orden que ocupan las tareas en la cola de preparados. De este modo, el rendimiento, tanto de las tareas locales como de las distribuidas, dependerá de la política seguida para ordenar las tareas dentro de la cola de preparados. Sin embargo, los estudios realizados en el capítulo anterior ponen de manifiesto que las tareas deben ser ordenadas de manera distinta en función del grado de ocupación de la memoria principal. Atendiendo a este doble comportamiento, CSC ordena las tareas, tanto locales como distribuidas, en la cola de preparados de acuerdo al algoritmo mostrado en la figura 3.3. Según nuestro modelo, el punto

óptimo para implementarlo será dentro de una función genérica denominada "*INSERTAR_RQ*", la cual se encargará de insertar las tareas listas para ejecución en la cola de preparados (RQ).

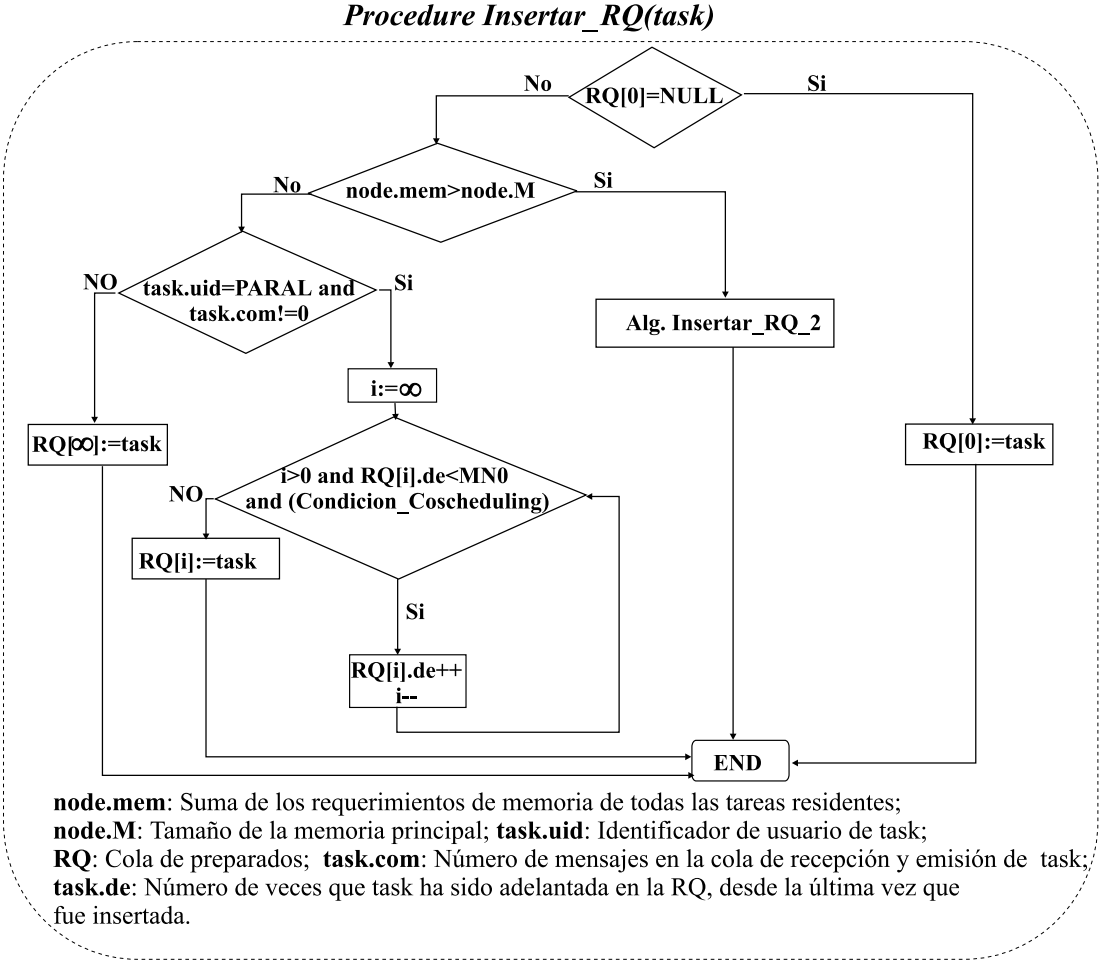

Figura 3.3: Algoritmo de inserción de tareas en las colas de preparados.

El algoritmo de la figura 3.3 muestra como, en aquellos nodos donde los requerimientos de memoria no superen el tamaño de la memoria principal ($node.mem < node.M$), las tareas son ordenadas en la cola de preparados de acuerdo con los requerimientos de comunicación de las mismas. En caso de que la memoria principal sea desbordada se aplicará el algoritmo, indicado como *Insertar_RQ_2*, descrito posteriormente en la sección 3.3.2.

La coplanificación de las tareas distribuidas de acuerdo con sus necesidades de comunicación es un factor imprescindible a tener en cuenta con

objeto de garantizar un rendimiento aceptable para las tareas distribuidas. El estado de arte de las técnicas de coscheduling (ver sección 1.3.2) revela que las técnicas basadas en la coplanificación de tareas en función de la recepción de mensajes, denominadas técnicas de coscheduling Dinámico, son las que mejor se adaptan a la variabilidad de los entornos cluster no dedicados. La técnica del coscheduling Dinámico fue definida en primer lugar por Sobalvarro [SW95]. En su propuesta, Sobalvarro introdujo dos versiones diferentes. En la primera de ellas, denominada "*always-schedule dynamic coscheduling*", la recepción de un mensaje siempre causa un cambio de contexto a favor de la tarea destino del mismo; mientras que en la segunda implementación, denominada "*equalizing dynamic coscheduling*" se limita el número de veces que una tarea paralela puede apropiarse de la CPU; de esta manera se evita una posible inanición de las tareas locales respecto de las tareas paralelas con elevada frecuencia de comunicación. Dado nuestro interés en preservar el rendimiento del usuario local nos centraremos en esta segunda propuesta.

El algoritmo original propuesto por Sobalvarro funciona de la siguiente manera: cuando el *node* recibe un mensaje para una $task_j$, asignada en dicho nodo y distinta de la tarea actualmente en ejecución ($task_i$), se producirá un cambio de contexto en favor de $task_j$ solamente en el caso de que se cumpla la desigualdad siguiente, la cual será denominada *condición de apropiación*:

$$ciclos\_exec_j + K\_SOBAL < ciclos\_exec_i \tag{3.5}$$

donde $ciclos\_exec_i$ es el número de ciclos de ejecución de $task_i$ transcurridos desde la última vez que la tarea fue insertada en la cola de preparados y K_SOBAL es la *constante de apropiación*. De esta manera, para valores pequeños de K_SOBAL, el algoritmo de Sobalvarro tenderá a favorecer a las tareas distribuidas con alta tasa de comunicación, mientras que para valores grandes tenderá a favorecer a las tareas con baja o nula tasa de comunicación, lo que comportará un bajo ratio de aciertos en el coscheduling de las tareas paralelas. Otro problema asociado con dicho algoritmo es que a medida que se aumenta el grado de multiprogramación de las tareas distribuidas, el rendimiento de las tareas locales se verá degradado, dado que un mayor número de tareas distribuidas competirán para apropiarse de la CPU. Este comportamiento hace que, en la práctica, el valor óptimo de la *constante de apropiación* sea muy difícil de determinar.

Los motivos anteriormente descritos nos llevaron a modificar el algoritmo original de Sobalvarro. En este nuevo algoritmo, la condición de apropiación se cambió con objeto de limitar el número de adelantamientos que puede sufrir una tarea local en la cola de preparados y, de esta manera, independizarlos del grado de multiprogramación de las tareas distribuidas. Asimismo,

mientras que el algoritmo original solamente tiene en cuenta las recepciones de mensajes a la hora de planificar las tareas, este nuevo algoritmo planifica en función tanto de los mensajes recibidos como de los enviados. De esta manera, la probabilidad de alcanzar coscheduling entre los procesos cooperantes aumenta. La figura 3.3 muestra el algoritmo propuesto.

Este algoritmo, cada vez que una nueva tarea distribuida (task.uid=PARAL) es insertada en la cola de preparados, comprueba si tiene algún mensaje en la cola de recepción o emisión (*task.com!=0*). En caso afirmativo, la tarea se apropiará de la CPU (*RQ[0]:=task*) siempre y cuando se cumpla la siguiente condición de apropiación:

$$(RQ[i].de < MNO)\ and\ (RQ[i].com < task.com) \tag{3.6}$$

Por tanto, las condiciones dinámicas aplicadas son dos:

- **El máximo número de adelantos sufridos (*MNO*)**. Un proceso no puede adelantar a otro si éste último ha alcanzado dicho máximo. De acuerdo con los resultados experimentales obtenidos [SGHL01a, SGHL01c], la constante MNO se ha tomado por defecto igual a dos. Un valor mayor que dos puede disminuir el ratio de aciertos del coscheduling por encima del *50 %,* mientras que un valor inferior a dos puede perjudicar en exceso el tiempo de respuesta de las tareas interactivas del usuario local. El apéndice B muestra un estudio sobre la variación del Slowdown de las tareas distribuidas y locales, con respecto al valor de la constante MNO y al grado de multiprogramación.

- **El número de mensajes recibidos y enviados (*task.com*)**. Esta segunda condición, también denominada *condición de cosheduling*, determina que un proceso puede adelantar a otro siempre y cuando tenga más mensajes pendientes en la cola de recepción y emisión que este último. De esta manera se prioriza aquellos procesos con mayor tasa de recepción/emisión de mensajes y que, por tanto, tienen una mayor probabilidad de que un mayor número de tareas cooperantes estén a la espera de su respuesta.

Un aspecto no reflejado en el algoritmo de la figura 3.3 es la estrategia tomada por los procesos que están a la espera de la recepción de un mensaje. Teniendo en cuenta el trabajo realizado por Anglano en esta línea [Ang00], el cual ha sido contrastado por diferentes autores [CAK+03, Sol02], se ha tomado una estrategia de *bloqueo inmediato*, caracterizada porque el proceso, en espera del mensaje, es bloqueado hasta la recepción del mensaje.

3.3.1. Coscheduling entre tareas distribuidas homogéneas

El algoritmo anteriormente descrito prioriza las tareas distribuidas de acuerdo con el número de mensajes en las colas de recepción y emisión de las tareas. Un problema intrínseco a esta técnica surge cuando se ejecutan concurrentemente varias copias de una misma aplicación distribuida, de manera que las tareas competidoras son homogéneas y por tanto, aumenta la probabilidad de que varias tareas en la cola de preparados posean el mismo número de mensajes en su respectivas colas. En esta situación, el planificador no es capaz de discernir entre tareas pertenecientes a diferentes trabajos, lo que comporta que la probabilidad de coscheduling entre tareas cooperantes pueda disminuir drásticamente.

Con objeto de contemplar esta situación, la condición de coscheduling original ($RQ[i].com < task.com$) fue modificada de acuerdo con la siguiente expresión:

$$RQ[i].com < task.com \; or \; (RQ[i].com = task.com \; and \; RQ[i].jid < task.jid). \tag{3.7}$$

La expresión 3.7 refleja la política evaluada por medio de la simulación en la sección 2.3.1, de manera que en caso de que varias tareas competidoras tengan el mismo número de mensajes se priorizará la tarea que tenga el mayor identificador ($task.jid$), es decir, aquella que haga menos tiempo que ha sido lanzada al sistema. De este modo, cada planificador será capaz de discernir entre tareas homogéneas pertenecientes a distintos trabajos y dado que todos los nodos aplicarán el mismo criterio, la probabilidad de coscheduling entre tareas cooperantes será incrementada.

3.3.2. Asignación de prioridades con restricciones de memoria

De acuerdo con la experimentación realizada en el capítulo anterior, cuando la suma de los requerimientos de memoria de las distintas tareas residentes en un determinado *node* ($node.mem$) supera el tamaño de su memoria principal ($node.M$) y, en consecuencia, se activa el mecanismo de *swapping*, el rendimiento de las técnicas de coscheduling basadas en la ocurrencia de eventos de comunicación disminuye drásticamente debido al efecto de los fallos de página. Asimismo, estos mismos resultados mostraron que una política basada en priorizar la ejecución de aquellas tareas con menor tasa de fallos de página puede disminuir, en promedio, la tasa de fallos de página global de un nodo.

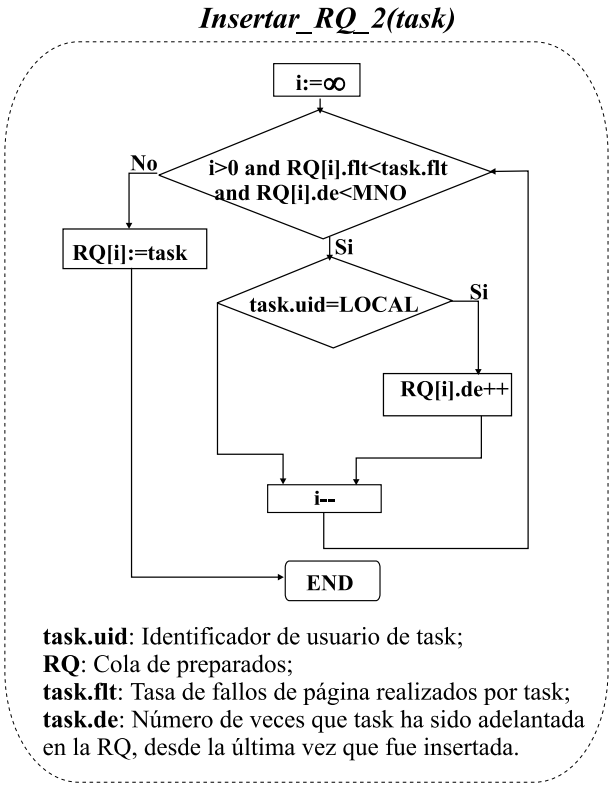

Figura 3.4: Algoritmo de ordenación de tareas de acuerdo con sus requerimientos de memoria.

Con este objetivo, el algoritmo de la figura 3.4 ordena las tareas en la cola de preparados de acuerdo con su tasa de fallos de página ($task.flt$). De este modo, aquellas tareas con menor requerimientos de memoria finalizarán antes su ejecución, de modo que el resto de tareas competidoras podrán hacer uso de los recursos liberados por las tareas finalizadas. Asimismo, al igual que se hacía en el algoritmo de coscheduling basado en eventos de comunicación, se ha implementado un mecanismo para evitar la inanición de aquellas tareas locales con altos requerimientos de memoria. Su objetivo es limitar el número máximo de adelantamientos sufridos por una tarea local dentro de la cola de preparados ($RQ[i].de < MNO$).

3.4. CoScheduling Cooperativo: Asignación de la Longitud del Quantum

Como se explicó en el capítulo de introducción, diferentes autores [YJ01, FPCF01] han puesto de manifiesto la sensibilidad de las técnicas de coscheduling tradicionales en general y, del coscheduling Dinámico en particular, con respecto a la longitud del quantum. En un entorno multiprogramado, un quantum excesivamente grande puede degradar el rendimiento de las técnicas de coscheduling. Este comportamiento es debido a que el tiempo que las tareas cooperantes deben esperar a ser planificadas en la cola de preparados, debido a que no cumplen la *condición de apropiación* del coscheduling, aumenta a medida que crecen las tareas competidoras, así como el quantum de tiempo asignado a cada tarea. Por otro lado, un quantum excesivamente corto también puede degradar el rendimiento de las tareas distribuidas debido a que éstas no tengan tiempo suficiente para poder establecer el coscheduling entre ellas.

Asimismo, el estudio realizado en el capítulo anterior revela que los entornos de planificación con longitudes de quantum grandes gestionan mejor la localidad de memoria de las tareas distribuidas, hecho que repercute en una disminución en el número de fallos de página, en situaciones donde la memoria principal es desbordada.

Con respecto a las tareas locales, un quantum excesivamente largo puede degradar el tiempo de respuesta de las tareas interactivas locales. Este hecho se pone de manifiesto en el trabajo realizado por Robert B. Miller [Mil68] en 1968, donde se fijan las recomendaciones en los tiempos de respuesta percibidos por un usuario interactivo:

- Dos décimas de segundo es el límite aproximado para hacer sentir al usuario que el sistema está reaccionando instantáneamente.
- Un segundo es el límite máximo para que el usuario piense que no hay interrupción, aunque éste se de cuenta de la demora.
- Diez segundos es el límite máximo para mantener la atención del usuario centrada en el dialogo.

Tomando como referencia estos umbrales, la investigación tradicional de los factores humanos sobre los tiempos de respuesta de sistemas interactivos muestran la necesidad de que éstos sean siempre inferiores a medio segundo [Nie00]. Siguiendo esta línea de trabajo y adoptando una política conservadora con respecto al rendimiento del usuario local interactivo, nuestro objetivo

será asegurar, al usuario local, un tiempo de respuesta siempre menor a medio segundo.

La interrelación entre los tres hechos anteriormente descritos (rendimiento del coscheduling, minimización de los fallos de página y tiempo de respuesta del usuario local) ponen de manifiesto la dificultad de fijar la longitud del quantum en un entorno cluster no dedicado multiprogramado. La figura 3.5 muestra el algoritmo propuesto para asignar el quantum teniendo en cuenta estos tres factores.

De acuerdo con el modelo descrito en la sección 2.1, este algoritmo será ejecutado por el planificador de cada nodo al final de cada época, es decir, en el instante en que todos los procesos listos para ejecución hayan consumido su quantum; y por tanto cada planificador tenga que reasignar, a todos los procesos residentes en dicho nodo, una nueva longitud de quantum.

Este algoritmo comprueba, en primer lugar, si la memoria ha sido desbordada. En caso afirmativo, se procederá a ejecutar el algoritmo descrito en la sección 3.4.1. En caso contrario, el algoritmo recorre todas las tareas residentes en la máquina, asignando un nuevo quantum, en función del estado de las tareas distribuidas (*task.state*) así como de la presencia del usuario local en dicha máquina ($node.LOCAL\_USER$).

Según la figura 3.5, el algoritmo detiene, es decir asigna un quantum igual a cero, aquellas tareas distribuidas que tienen alguna tarea cooperante que ha sido parada en un nodo remoto y, por consiguiente, tienen un estado de STOP (ver traza azul). Asimismo, el algoritmo detiene aquella tarea distribuida que ha sido etiquetada como *stoptask* por el algoritmo de gestión de memoria explicado en la sección anterior. En este sentido cabe destacar que el algoritmo, al inicio del mismo, comprueba la viabilidad de reanudar *stoptask;* siendo ésta reanudada solamente en el caso de que los requerimientos de memoria (*node.mem*) hayan disminuido por debajo de un cierto umbral definido como MEM_MIN. Acorde con la experimentación realizada (ver sección B.2 del apéndice B), este umbral ha sido tomado igual al *95 %* del valor de la memoria principal.

En caso de que la máquina esté ocupada por el usuario local, el algoritmo distingue entre si la tarea pertenece al usuario local o bien al paralelo. En caso de que la tarea sea del usuario local ($task.uid!=PARAL$) se le asignará una longitud de quantum, para la siguiente época ($task.q_{n+1}$), igual al valor de quantum fijado por defecto por el planificador original ($DEF\_QUANTUM$), mientras que si la tarea es paralela se le asignará un quantum proporcional al porcentaje de recursos de cómputo asignados a las tareas paralelas (marcado en el algoritmo en color rojo). Este modo de proceder comporta que el porcentaje de tiempo de cada época consumido por las tareas paralelas crecerá a medida que aumente tanto el grado de multiprogramación de las tareas

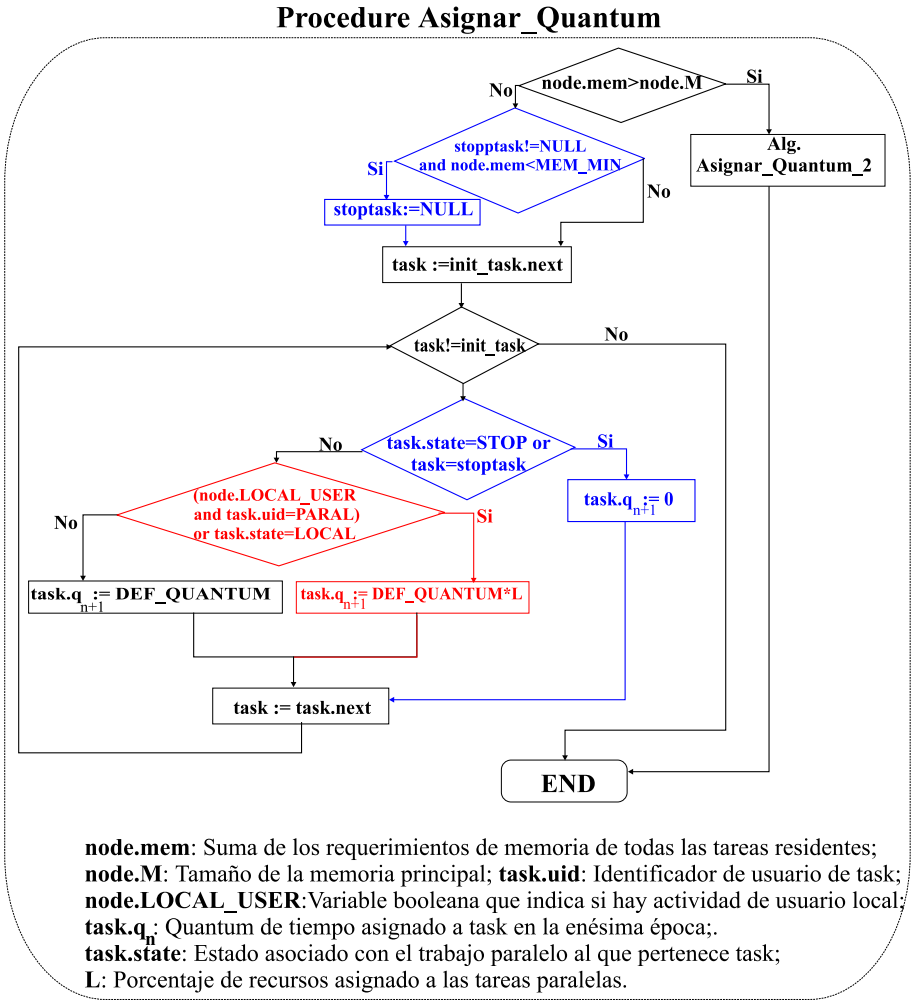

Figura 3.5: Algoritmo de asignación de quantum.

paralelas ($node.MPL$), como el valor del porcentaje de recursos asignados (L). Por tanto, un equilibrio entre el valor del porcentaje L y el grado de multiprogramación $node.MPL$ deberá ser tenido en consideración. En este sentido, los valores extremos tanto de $node.MPL$ como de L vendrán dados por la siguiente condición:

$$node.MPL * DEF\_QUANTUM * L \leq 400msg. \tag{3.8}$$

De este modo, en el peor de los casos con respecto al usuario local, se limita el tiempo de CPU consumido por las tareas paralelas en cada época a $400msg$; valor por debajo del cual el usuario interactivo no percibe una

ralentización de su trabajo [Nie00]. Por tanto, teniendo en cuenta que la mayoría de s.o. UNIX actuales tienen un quantum por defecto alrededor de 200ms[2], el producto $node.MPL * L$ deberá ser siempre menor que 2.

Asimismo en los nodos sin usuarios locales, el valor del quantum para aquellas tareas distribuidas que tienen un estado igual a $LOCAL$[3] será decrementado proporcionalmente al porcentaje de recursos de cómputo asignados a las tareas paralelas. De este modo, todas las tareas pertenecientes a una misma aplicación distribuida tendrán asignadas la misma longitud de quantum a lo largo del cluster.

3.4.1. Asignación de quantum en entornos con altos requerimientos de memoria

El algoritmo presentado en esta sección es aplicado en aquellos nodos donde la memoria ha sido desbordada ($node.mem > node.M$), con el objetivo de explotar la localidad de memoria de cada proceso, mediante un incremento de la longitud del quantum. Este algoritmo, mostrado en la figura 3.6, es una extensión del presentado en la sección 3.4. Tal como se puede ver en la figura 3.6, este nuevo algoritmo tiene dos maneras bien distintas de trabajar, dependiendo de si está presente el usuario local ($node.LOCAL\_USER$). En función de este parámetro se distinguen los siguientes modos de trabajo:

1. **Swapping con un usuario local:** En estas condiciones, el algoritmo procederá a rebajar la memoria consumida por la ejecución de tareas distribuidas, siempre y cuando dicho desbordamiento sea debido a que las tareas distribuidas hayan sobrepasado el límite de memoria concedido a las mismas (ver sección 3.2). Con este fin, el algoritmo procederá a detener, asignando un quantum de tiempo igual a cero ($task.q_{n+1} := 0$), y desalojar de memoria a la tarea distribuida, que bien haya sido notificada por el algoritmo de gestión de memoria como *stoptask* o bien tenga su estado igual a *STOP*. De este modo se ajusta dinámicamente el grado de multiprogramación de acuerdo con los requerimientos de memoria. El proceso parado no será reanudado hasta que no se haya ido el usuario local, o bien hasta que los requerimientos de memoria totales disminuyan por debajo del umbral MEM_MIN $(= 95\,\% * node.M)$, definido en el algoritmo 3.5. El resto de tareas paralelas tendrán asignado un quantum proporcional al porcentaje L,

[2]Por ejemplo, LINUX (v.2.2) tiene una longitud de quantum igual a 210ms.

[3]Una tarea distribuida tiene un estado LOCAL cuando alguna tarea cooperante, ejecutándose en un nodo remoto, tiene asignado un quantum decrementado debido a la presencia de un usuario local en dicho nodo.

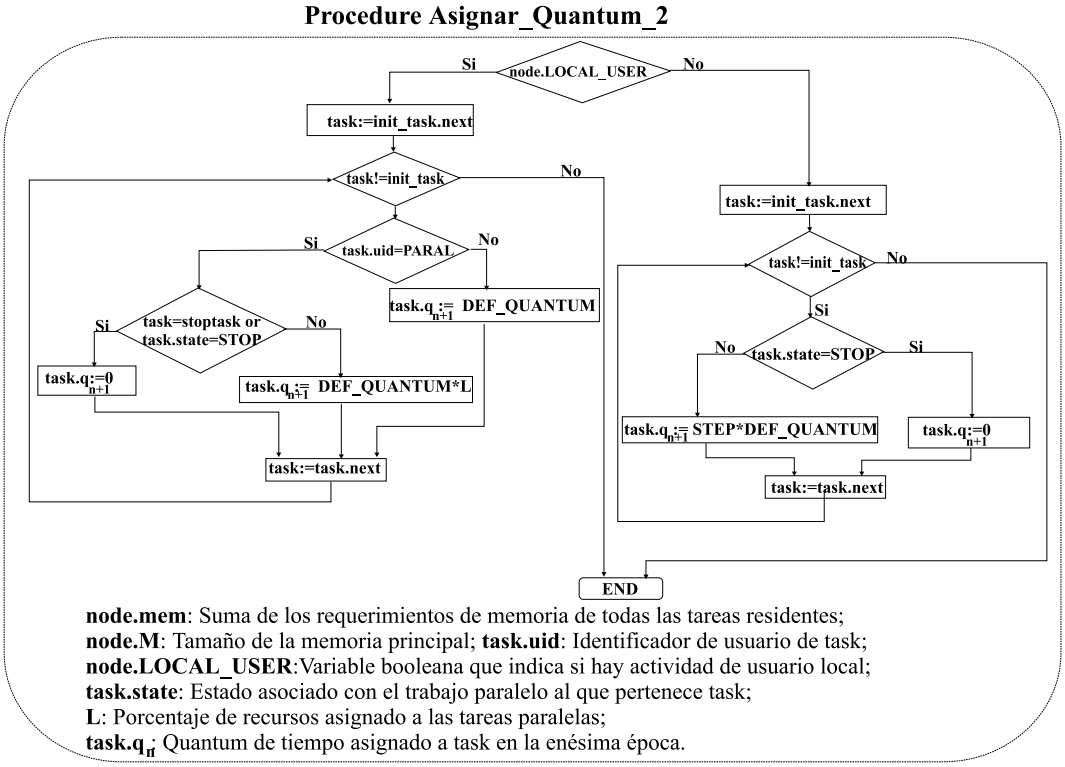

Figura 3.6: Algoritmo de asignación de quantum en entornos con altos requerimientos de memoria.

mientras que las locales tendrán asignado el quantum establecido por defecto.

2. **Swapping sin usuario local**: Experimentalmente se comprobó, en el capítulo anterior, como un incremento de la longitud del quantum puede contribuir a reducir el número de fallos de página, dado que cada proceso puede cargar mejor en memoria su *working set*. Con este fin, el nuevo quantum de tiempo asignado a todos los procesos residentes en dicho nodo, es incrementado, con respecto a la longitud de quantum asignado por defecto por el s.o, en un factor constante denominado *STEP*. En el capítulo de experimentación (capítulo 5) se comprobará como con un valor de $STEP = 5$, el número de fallos de página se reduce considerablemente y, por consiguiente, el rendimiento global de las aplicaciones distribuidas es mejorado.

3.4.2. Heterogeneidad del procesador

Con el objetivo de cuantificar la heterogeneidad de los diferentes procesadores que forman una NOW se ha utilizado el *Power Weight* ($node_i.W$), métrica que da la potencia relativa del procesador del $node_i$ con respecto al procesador más rápido presente en el cluster ($node_{max}$). De acuerdo con la definición dada por Zhang et al [DZ97], *Power Weight* se define como:

$$node_i.W(App) = \frac{min_{j=1}^{n}\{T(App, node_j)\}}{T(App, node_i)}, \tag{3.9}$$

siendo n el número de nodos del cluster y $T(App, node_i)$, el tiempo de ejecución de una determinada aplicación (App) en el $node_i$. Sin embargo, de acuerdo con los trabajos de los mismo autores, el *Power Weight* puede considerarse independiente del tipo de aplicación, siempre y cuando ésta encaje en memoria. De acuerdo con estos resultados, el *Power Weight de un* $node_i$ será indicado como $node_i.W$.

Atendiendo al grado de heterogeneidad de cada nodo, el quantum de tiempo asignado, por defecto, por el planificador de cada $node_i$ es redefinido de acuerdo con la siguiente expresión:

$$node_i.DEF\_QUANTUM = \frac{DEF\_QUANTUM}{node_i.W}, \tag{3.10}$$

donde $DEF\_QUANTUM$ es el quantum por defecto asignado por el s.o. y $node_i.W$ es el *Power Weight* del $node_i$.

Como se describió en el modelo básico del cluster (sección 2.1), la duración máxima de cada época del planificador vendrá dada por la siguiente igualdad:

$$node_i.epoca_k = node_i. \sum_{j=1}^{node_i.MPL\_TOTAL_k} task_j.q_k, \tag{3.11}$$

donde $node_i.MPL\_TOTAL_k$ es el nivel de multiprogramación del $node_i$ durante la késima época ($epoca_k$), contando tanto las tareas locales como las paralelas, y $tas_j.q_k$ es la longitud del quantum asignado por el planificador del $node_i$ a $task_j$ durante la késima época. Teniendo en cuenta esta relación, la duración de cada época dependerá tanto del tamaño del quantum asignado a cada una de las tareas que se ejecutan en dicho nodo, como del número de tareas que se ejecutan en el mismo.

La situación ideal sería que la duración de cada época fuese uniforme a lo largo del cluster, independientemente de la potencia del procesador correspondiente. Alcanzar este objetivo significa que se cumpla la siguiente igualdad:

$$node_i.epoca_k = node_{max}.epoca_k,\ \forall i \in \{1, ..., n\}, \tag{3.12}$$

siendo n el número de nodos del cluster y $node_{max}$ el nodo más rápido presente en el cluster. Atendiendo a las ecuaciones 3.11 y 3.10, la relación 3.12 puede ser reescrita, de acuerdo con la siguiente expresión:

$$node_i.MPL\_TOTAL_k = node_{max}.MPL\_TOTAL_k * node_i.W,\ \forall i \in \{1, ..., n\}, \tag{3.13}$$

siendo n el número de nodos del cluster. La relación 3.13 demuestra que el grado de multiprogramación de cada nodo debe fijarse también en función de su *Power Weight* y del grado de multiprogramación asociado con el nodo con un mayor *Power Weight* ($node_{max}$). Cabe remarcar que, en un entorno real productivo, el grado de multiprogramación será totalmente dinámico, al depender en gran medida del comportamiento del usuario local de cada uno de los nodos del cluster. Sin embargo, con objeto de intentar aproximarnos a dicha situación ideal, la relación 3.13 debería ser tenida en cuenta a la hora de fijar el grado de multiprogramación de las tareas paralelas en entornos heterogéneos. De esta manera, junto con el algoritmo de balanceo de recursos, explicado en la sección 3.5, las tareas cooperantes que forman una misma aplicación podrán progresar de una manera uniforme a lo largo de cada época del planificador, independientemente de la potencia del procesador del nodo en el cual han sido mapeadas.

3.5. CoScheduling Cooperativo: Asignación Uniforme de Recursos de Cómputo a lo Largo del Cluster

El objetivo primordial de un algoritmo de coscheduling es mantener la coordinación entre todas las tareas distribuidas pertenecientes a un mismo trabajo paralelo (*tareas cooperantes*) que se están ejecutando a lo largo del cluster. Esta coordinación comporta que todas las tareas cooperantes sean planificadas simultáneamente con una prioridad de asignación de los recursos de cómputo (CPU y memoria) uniforme a lo largo del cluster. Sin embargo, la necesidad de preservar los recursos de cómputo para las tareas locales, en aquellos nodos donde el usuario local está presente, puede provocar que los recursos asignados a las tareas paralelas no estén balanceados a lo largo del cluster, de manera que tareas cooperantes de una misma aplicación distribuida tengan diferentes prioridades de asignación de los recursos de cóm-

puto disponibles en el nodo donde se están ejecutando. Esta asignación no uniforme de los recursos puede comportar que las tareas cooperantes progresen asíncronamente a lo largo del cluster y, en consecuencia, disminuya el rendimiento global de dichas aplicaciones.

Esta asignación podría mejorarse si cada nodo conociese el estado de aquellas tareas cooperantes que se ejecutan en nodos remotos. De este modo, todos los nodos podrían asignar la misma prioridad de asignación de los recursos a todas las tareas que forman una misma aplicación distribuida. Sin embargo, este objetivo no puede ser alcanzado solamente a partir del análisis de los eventos locales, como se realizaba en los algoritmos precedentes de coscheduling. Este nuevo objetivo comporta que los nodos remotos se intercambien información, entre ellos, acerca de como distribuyen sus recursos disponibles entre las tareas distribuidas. No obstante, esta distribución de recursos fluctúa en función de la ocurrencia de determinados eventos, como por ejemplo, la presencia (no presencia) de un usuario local, el grado de multiprogramación de cada nodo, el estado del subsistema de memoria, etc... Por esta razón, nuestra propuesta se basa en que cada nodo notifique la ocurrencia de aquellos eventos que comportan una limitación en los recursos de cómputo para las tareas paralelas residentes en dicho nodo. Esta notificación será enviada a todos aquellos nodos que disponen de tareas cooperantes con las del nodo en cuestión (*nodos cooperantes*). Sin embargo, un número elevado de eventos intercambiados puede incrementar ostensiblemente la complejidad del sistema y, por consiguiente, limitar la escalabilidad del mismo. Por este motivo, solamente se notificarán los siguientes cuatro eventos:

- $LOCAL$. Este evento será enviado por el nodo bajo dos circunstancias distintas:
  - Cuando el usuario local se incorpore a trabajar en dicho nodo. De acuerdo con nuestro modelo, este evento se notificará cuando el valor de la variable booleana local $LOCAL\_USER$ realice una transición de 0 a 1 ($0 \rightarrow 1$).
  - Cuando habiendo un usuario local en la máquina ($LOCAL\_USER = 1$), se lance una nueva aplicación paralela y, por consiguiente, se incremente el grado de multiprogramación paralelo ($\triangle MPL$).
- NO_LOCAL. Este evento será disparado en dos circunstancias diferentes:
  - Cuando el usuario local abandone el nodo, es decir, cuando el valor de la variable booleana local $LOCAL\_USER$ realice una transición de 1 a 0 ($1 \rightarrow 0$).

- En el caso de que habiendo un usuario local ($LOCAL\_USER = 1$) finalice una tarea distribuida y, por consiguiente, se decremente el grado de multiprogramación ($\bigtriangledown MPL$).

- *STOP*. Este evento se producirá cuando una determinada tarea paralela sea detenida (*task*) por el módulo de gestión de memoria debido a que la memoria principal se haya desbordado (ver algoritmo 3.2). Por tanto, este evento se notificará cuando el valor del apuntador local *stoptask* realice una transición de NULL a task ($NULL \rightarrow task$).

- *RESUME*. Este evento se producirá cuando la tarea detenida (*task*) sea reanudada por el planificador debido a que los requerimientos de memoria en dicho nodo hayan descendido por debajo del umbral *MEM_MIN* (ver algoritmo 3.5). Por tanto, este evento se disparará cuando el valor del apuntador *stoptask* realice una transición de *task* a NULL ($task \rightarrow NULL$).

El algoritmo de la figura 3.7 ilustra el procedimiento realizado para el envío de los eventos anteriormente descritos. De acuerdo con el modelo descrito en la sección 2.1, este algoritmo será ejecutado, por cada uno de los nodos que componen nuestro cluster, al final de cada época. En el caso de los eventos *LOCAL* y *NO_LOCAL*, el algoritmo recorre cada una de las tareas distribuidas que se están ejecutando en el correspondiente nodo y envía, para cada una de ellas, una notificación de dicho evento a todos aquellos nodos donde se encuentran ejecutando las respectivas tareas cooperantes. Cabe recordar que de acuerdo con nuestro modelo, cada nodo dispone de una lista denominada *cooperating*, que contiene las direcciones de todos los nodos donde están ejecutándose las tareas cooperantes de una determinada tarea *task*. En cambio, para el caso de los eventos *STOP* y *RESUME,* solamente se enviará la notificación del evento a aquellas tareas cooperantes de la tarea detenida (*stoptask*) o reanudada.

Cada notificación de un evento comporta el envío de los siguientes dos campos de información:

- *event*. Corresponde a un código asociado con el evento notificado. Los valores de este código son los siguientes: *LOCAL, NO_LOCAL, STOP y RESUME.*

- *task.jid*. Corresponde al identificador de la aplicación distribuida a la cual pertenece la tarea objeto de la notificación.

La recepción de cualquiera de los eventos anteriores por parte de un determinado nodo, provocará la ejecución del algoritmo que se muestra en la

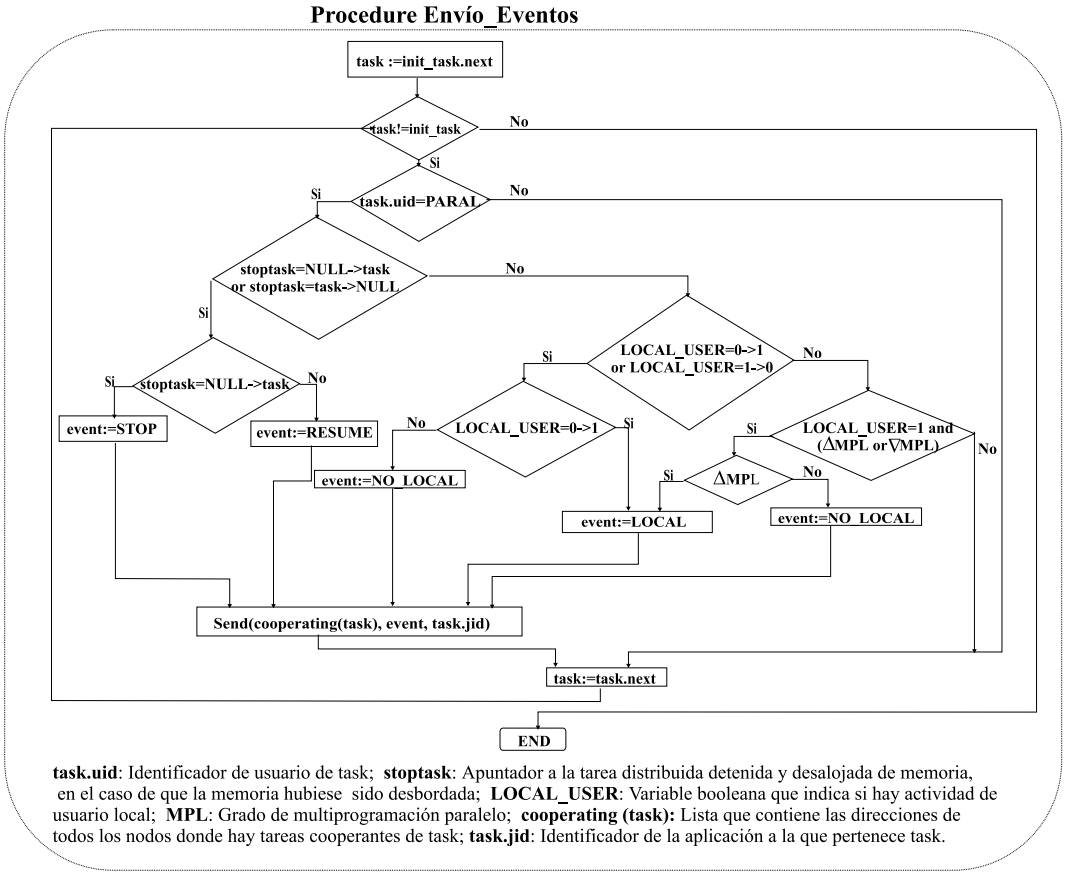

Figura 3.7: Algoritmo de envío de eventos.

figura 3.8. Este algoritmo busca, en primer lugar, aquella tarea (*task*) que tiene el mismo identificador (*task.jid*) que el enviado por el nodo remoto (*jid_remote*). A continuación, el algoritmo procede a reasignar el estado de la tarea notificada, de acuerdo con el tipo de evento recibido. De este modo, el resto de bloques que configuran CSC serán capaces de reasignar los correspondientes recursos que gestionan de un modo coordinado con sus nodos cooperantes.

En el caso de que el evento recibido sea *LOCAL*, el algoritmo le asignará a la tarea notificada, siempre y cuando sea distinta de *stoptask*, el estado *LOCAL*. En el caso de un evento *NO_LOCAL*, el estado será cambiado a *NULL* con objeto de que el algoritmo de asignación de quantum (ver sección 3.4) reasigne el quantum correspondiente. Cabe destacar que esta reasignación solamente se realizará en el caso de que se hayan recibido el mismo número de eventos *LOCAL* como *NO_LOCAL*. Esta comparación se realiza por medio

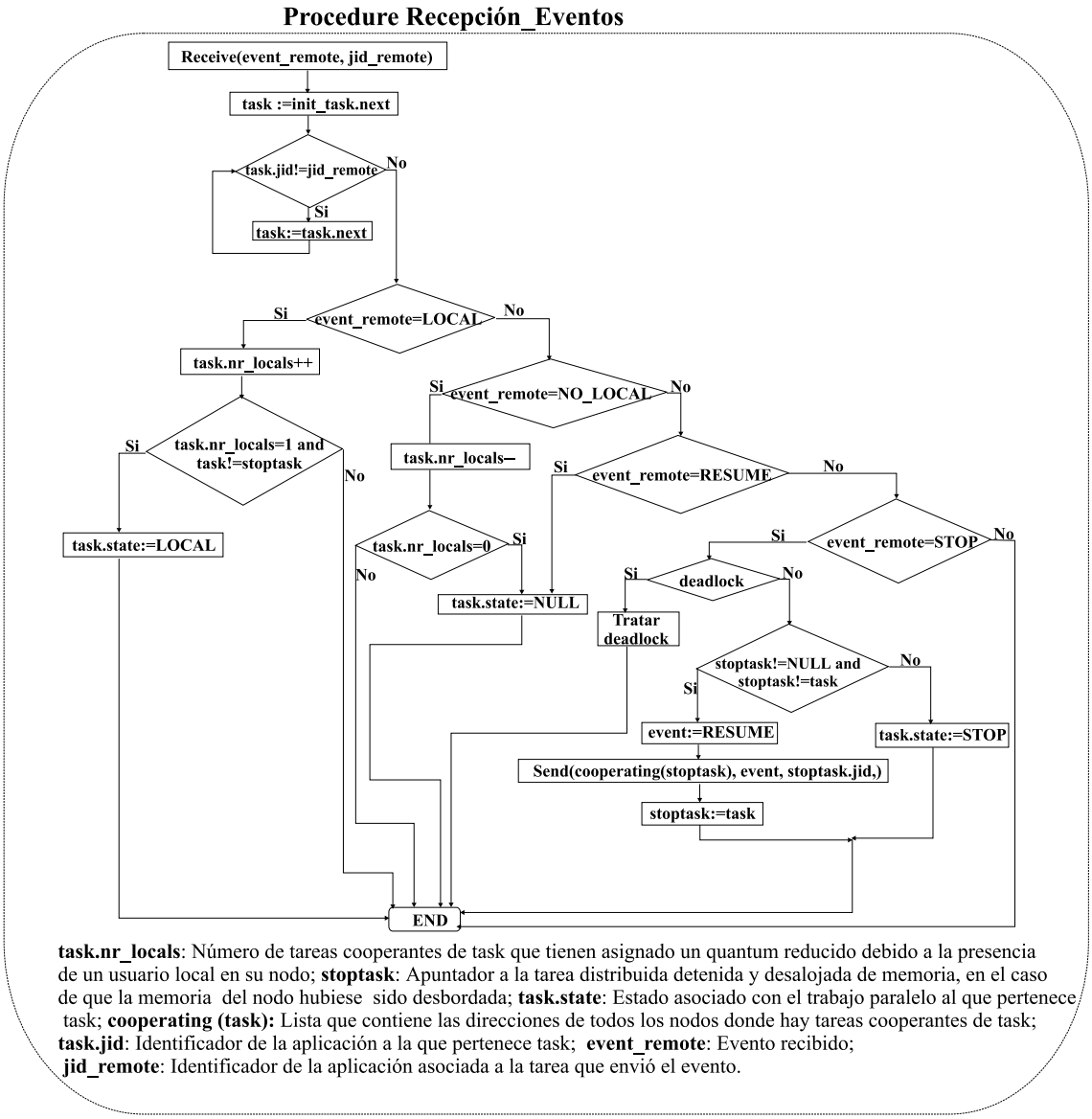

Figura 3.8: Algoritmo de recepción de eventos.

de una nueva variable, asociada a cada tarea, denominada *nr_locals*, la cual es incrementada cada vez que se recibe un evento *LOCAL* y decrementada cada vez que se recibe un evento *NO_LOCAL*. Por tanto, cuando el valor de dicha variable sea igual a cero (*task.nr_locals=0*) significará que ninguna tarea cooperante de *task* tiene reducidos sus recursos debido a la presencia de un usuario local en su máquina y por tanto, ya pueden ser incrementados

los recursos de todas las tareas que forman parte de la misma aplicación a la cual pertenece *task*.

En el caso de la recepción de un evento *RESUME*, el algoritmo procederá a reanudar la tarea detenida (*stoptask*), mediante la asignación del estado de *NULL* y la posterior aplicación de los algoritmos 3.5 y 3.6 por parte del bloque de gestión del quantum de CSC.

Finalmente, el proceso llevado a cabo para la recepción de un evento de *STOP* es un tanto más complejo debido a la posibilidad de que se produzca una situación de *deadlock* entre el nodo emisor y el nodo receptor del mensaje. La situación de *deadlock* ocurre cuando dos nodos, de manera simultánea, envían un evento de *STOP* asociado a dos trabajos distintos, tal como ilustra la figura 3.9. En esta figura se observa como el *nodo1* envía un evento de *STOP* asociado a todas las tareas que pertenecen a *J1* (indicado mediante flechas continuas), mientras que, al mismo tiempo, el *nodo4* envía también un mismo evento de *STOP* a todas las tareas de *J2* (indicado mediante flechas discontinuas). Esta situación provocará que ambos nodos (el *nodo1* y el *nodo4)* se vayan intercambiando ininterrumpidamente eventos de *STOP* asociados a *J1* y a *J2*, de manera que vayan permutando las correspondientes tareas detenidas. Por tanto, en función de la ocurrencia del *deadlock* se actuará de dos modos bien distintos:

- En el caso de que no ocurra *deadlock*, el algoritmo procederá a detener la tarea notificada. En este caso se deben distinguir dos situaciones distintas:

  1. Si en dicho nodo ya hubiese una tarea detenida (*stoptask*!=*NULL*) por parte del gestor de memoria, el algoritmo de recepción reanudará la antigua tarea detenida y procederá a detener la nueva tarea notificada. Cabe decir que esta situación solamente puede producirse en entornos donde coexistan aplicaciones distribuidas formadas por distinto número de tareas.
  2. En caso de que no hubiese ninguna tarea detenida, el estado de la nueva tarea notificada será cambiado a *STOP*, lo que comportará que el bloque de gestión del quantum le asigne una longitud igual a cero. Asimismo, en caso de que en un futuro se produzca desbordamiento de memoria en dicho nodo, dicha tarea será escogida como tarea *swaptask* por el módulo de gestión de memoria.

- La situación de *deadlock* es detectada, por el algoritmo de recepción mostrado en la figura 3.8, mediante la evaluación de la siguiente condi-

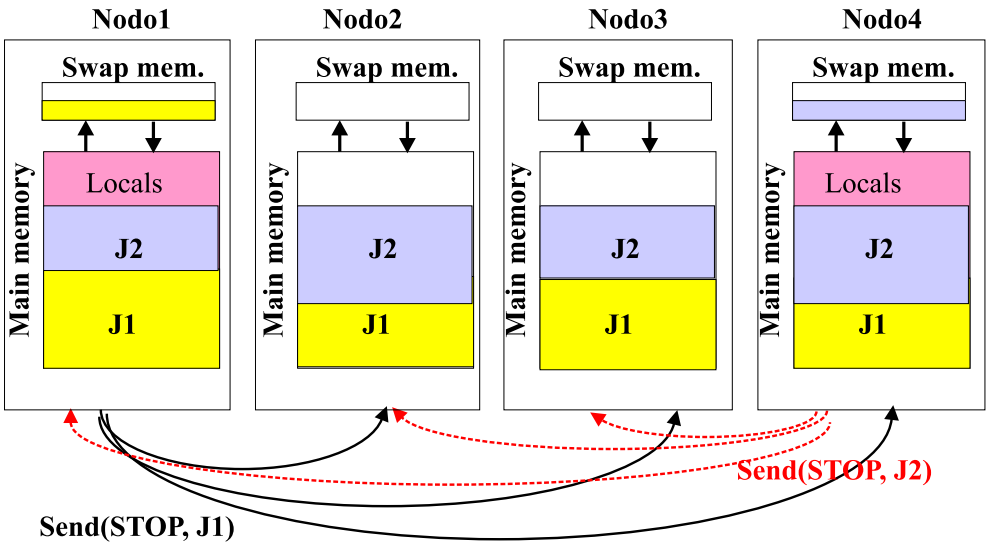

Figura 3.9: Situación de *deadlock*.

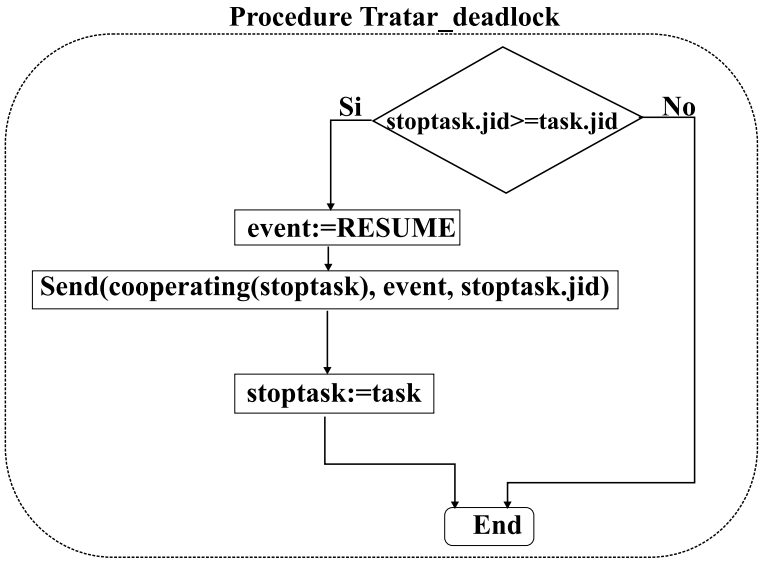

Figura 3.10: Rutina de tratamiento del *deadlock*.

ción (indicada en dicho algoritmo mediante la etiqueta *deadlock*):

$$(addressnode_{send} \in cooperating(stoptask)) and (stoptask.jid! = jid\_remote). \tag{3.14}$$

En el primer término de la expresión se comprueba si la dirección de

la máquina que realizó el envío del evento recibido ($addressnode_{send}$) pertenece al conjunto de máquinas donde hay tareas cooperantes de la tarea detenida en el nodo receptor del mensaje (*cooperating(stoptask)*). El segundo término evalúa si las dos máquinas, la receptora y la emisora del mensaje, han detenido dos tareas pertenecientes a distintos trabajos paralelos. En el caso de que se cumplan ambas condiciones nos encontraremos ante una situación de *deadlock* que será tratada de acuerdo con el algoritmo mostrado en la figura 3.10. El algoritmo de tratamiento de *deadlock* procederá a detener a aquella tarea con el identificador más pequeño, es decir aquella tarea que hace más tiempo que fue lanzada al sistema. Por tanto, en caso de que la tarea a detener sea la *stoptask*, no se hace nada puesto que ya está detenida, mientras que en el caso de que la nueva tarea notificada por el evento remoto tenga el menor identificador se tendrá que reanudar la *stoptask* anterior y detener la nueva tarea notificada.

3.6. Evaluación por Medio de Simulación de la Técnica de CoScheduling Cooperativo

En esta sección se analizará, por medio de la simulación, el rendimiento de la técnica de CSC bajo diferentes escenarios. En cada uno de estos escenarios se ha comparado la técnica de *CSC,* con un porcentaje fijo de recursos reservado a las tareas paralelas (L) del *50 %*, con respecto al *coscheduling Dinámico* tradicional y a un esquema de planificación no coordinado, como es el *Round-Robin*, tanto en lo que respecta al rendimiento de las tareas locales como al rendimiento de las tareas distribuidas.

Los tres gráficos mostrados en la figura 3.11(izq.) muestran el slowdown, correlación y overhead de las tareas distribuidas en un entorno donde todas las tareas residentes en un nodo encajan en memoria principal. En este escenario se han realizado diferentes simulaciones variando el grado de multiprogramación (MPL_TOTAL), entre uno y cinco, y fijando el número de trabajos locales generados (plt = 0,3). En general, los resultados obtenidos reflejan como la técnica del coscheduling Dinámico obtiene el mejor resultado para grados de multiprogramación bajos, mientras que la técnica de Round-Robin obtiene unos resultados muy pobres debido a la ausencia total de coordinación entre los nodos. Respecto a la técnica del CSC, el análisis del slowdown muestra como el rendimiento de CSC se va aproximando al del coscheduling Dinámico a medida que se aumenta el grado de multiprogramación, hasta superarlo para un grado de multiprogramación superior a

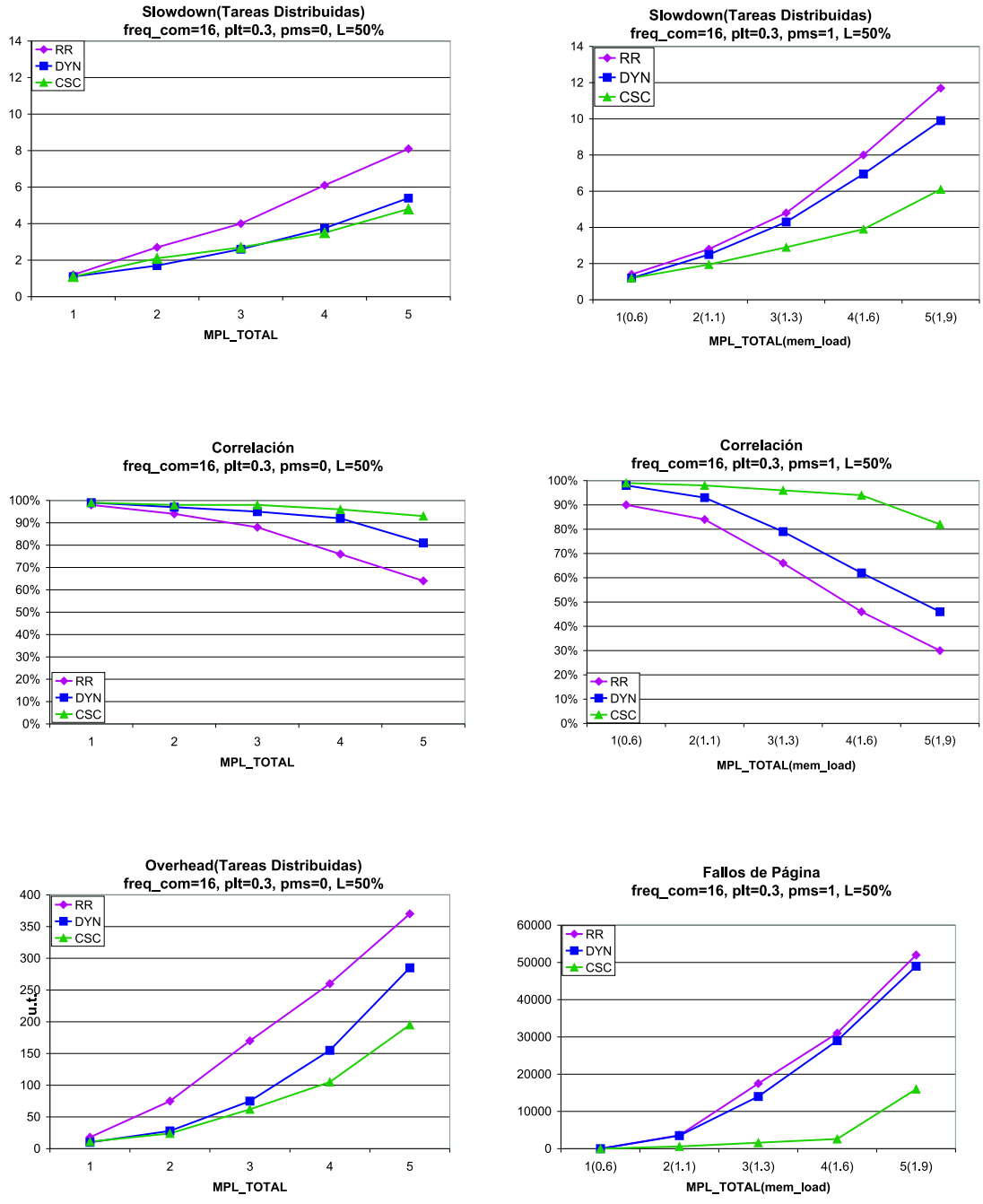

Figura 3.11: Rendimiento de las tareas distribuidas en condiciones de baja demanda de memoria (izquierda) y de alta demanda de memoria (derecha).

cuatro. Este comportamiento de CSC es debido a que al aumentar el grado de multiprogramación y por consiguiente el número de tareas distribuidas con respecto de las locales, la eficiencia en la asignación de recursos entre las tareas distribuidas por parte de CSC se hace más patente y, en consecuencia, aunque CSC asigne una menor cantidad de recursos (L=*50 %*), éstos son mejor gestionados a lo largo del cluster debido al gestor de recursos remotos incorporado en CSC. Esta eficiencia en la asignación de los recursos se refleja en las métricas de correlación y overhead obtenidas por CSC. Con respecto al overhead obtenido, la diferencia entre el overhead introducido por CSC y el Dinámico aumenta considerablemente con respecto al grado de multiprogramación. Esta diferencia es debida a que el gestor de quantum de CSC asigna un quantum inferior (=$DEF\_QUANTUM$*L *con* L=*0,5*) a las tareas distribuidas que el Dinámico (=$DEF\_QUANTUM$), de manera que en caso de que las tareas cooperantes no puedan ser planificadas debido a la propia *condición de apropiación* (ver sección 3.3), el tiempo que deberán esperar en la cola de preparados bajo la política CSC será menor, hecho que comporta una disminución del overhead.

La figura 3.11(der.) muestra los resultados obtenidos en un entorno con altos requerimientos de memoria ($pms = 1$). Entre paréntesis, en el eje de abscisas, se muestra la métrica mem_load obtenida para cada una de las simulaciones realizadas. Los resultados obtenidos en este entorno reflejan la capacidad que tiene CSC de gestionar eficientemente los recursos de memoria, debido al análisis, realizado por éste, de los eventos de memoria. Como consecuencia, CSC obtiene una tasa de fallos de página mucho más pequeña que el coscheduling Dinámico o Round-Robin. Estos buenos resultados obtenidos por CSC son debidos a dos razones distintas: en primer lugar, CSC detecta que el swapping es activado y, en consecuencia, detiene la aplicación distribuida con mayores requerimientos de memoria, disminuyendo, por consiguiente, el grado de multiprogramación en una unidad; en segundo lugar, esta mejora también es debida al incremento en la longitud del quantum ($STEP = 4$) que realiza el gestor del quantum de CSC en entornos con la memoria desbordada y sin usuarios locales. Esta disminución en el número de fallos de página se traduce en un menor slowdown y correlación. No obstante, para un grado de multiprogramación igual a cinco se observa como no es suficiente detener una única aplicación distribuida y, en consecuencia, el slowdown obtenido con CSC se incrementa drásticamente. Cabe resaltar que si CSC detuviese más de un trabajo distribuido, la complejidad asociada con el algoritmo de balanceo de recursos aumentaría considerablemente y, en consecuencia, la ganancia obtenida podría verse diluida por el overhead asociado con el balanceo de recursos a lo largo del cluster.

Un aspecto a resaltar, en los resultados obtenidos por CSC en ambos

entornos, es la elevada correlación obtenida por éste, hecho que indica la convergencia de de los algoritmos de emisión y recepción de eventos descritos en la sección 3.5.

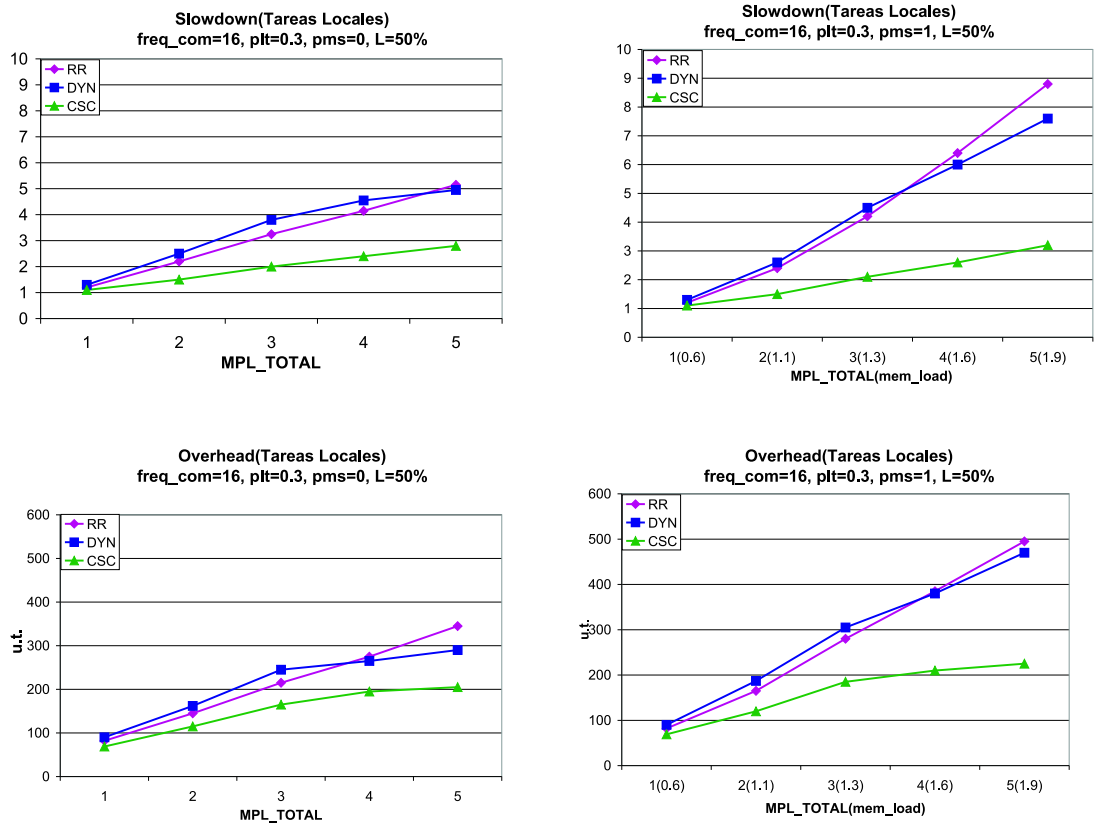

Figura 3.12: Slowdown y overhead de las tareas locales al ser ejecutadas con una carga distribuida con bajos (izquierda) y altos requerimientos de memoria (derecha).

En los mismos entornos mostrados anteriormente se analizó el rendimiento de las tareas locales. El slowdown y overhead obtenidos para ambos entornos son mostrados en la figura 3.12. Los dos gráficos de la izquierda corresponden a los resultados obtenidos cuando las tareas encajan en memoria, mientras que los gráficos de la derecha corresponden al caso en el que las tareas no encajan en memoria.

El slowdown de las tareas locales con bajos requerimientos de memoria muestran como CSC, de acuerdo con sus objetivos, es capaz de preservar el rendimiento de las tareas locales. Estos buenos resultados de CSC son debidos a la interacción de las dos técnicas utilizadas por CSC para lograr

este propósito: por un lado, CSC limita los recursos, tanto de CPU como de memoria, asignados a las tareas distribuidas. En concreto, estas pruebas fueron obtenidas para un porcentaje de recursos asignados igual al *50 %* y, por otro lado, CSC, al igual que el Dinámico, limita el número máximo de adelantos sufridos (MNO) por las tareas locales. Debido a que ambas aplican esta última técnica, las tendencias de las gráficas del CSC y Dinámico son similares. Sin embargo, el importante decalaje entre ambas gráficas demuestra la necesidad de limitar no solo los adelantos sufridos por las tareas locales, si no también la cantidad máxima de recursos asignados a las tareas distribuidas. El análisis del overhead refleja perfectamente como para un grado de multiprogramación superior a tres, la pendiente del overhead se suaviza, tanto para el Dinámico como para el CSC, de modo que por encima de un determinado valor de MPL_TOTAL, el overhead prácticamente se mantiene constante, garantizando de esta manera un buen tiempo de respuesta para el usuario interactivo local. Este efecto es debido a que tanto CSC como el Dinámico limitan a dos el máximo número de adelantos sufridos por las tareas locales.

La figura 3.12(der.) muestra el slowdown y overhead de las tareas locales al ser ejecutadas junto con trabajos distribuidos con altos requerimientos de memoria. El análisis de estos resultados muestran como CSC, de acuerdo con sus objetivos, consigue mantener el slowdown por debajo de 2,5 y el overhead por debajo de $220 u.t.$ para $MPL\_TOTAL < 5$, debido tanto a la aplicación del *algoritmo de asignación de quantum*, explicado en la sección 3.4, como de la aplicación del *algoritmo de gestión de memoria* descrito en la sección 3.2. La aplicación de ambos algoritmos permite adaptar los recursos de cómputo a las necesidades del usuario local, manteniendo el porcentaje de memoria reservado para el usuario local, así como mantener la progresión de aquellas tareas distribuidas con menor requerimiento de memoria. Los malos resultados obtenidos, tanto por el Round-Robin como por el Dinámico, son debidos a que estas dos políticas no discriminan a la hora de seleccionar la tarea para intercambiar páginas, con lo que las tareas locales sufren directamente el efecto de la paginación producido por los elevados requerimientos de las tareas distribuidas.

Los resultados mostrados a lo largo de esta sección ponen de manifiesto el potencial de la técnica de CoScheduling Cooperativo para este tipo de entornos. Sin embargo, la gran cantidad de parámetros que influyen en el rendimiento de CSC comporta la necesidad de analizar su rendimiento en un entorno cluster real. Con este fin, se ha procedido a su implementación en un entorno PVM_LINUX.